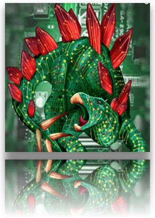
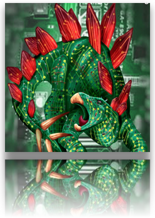


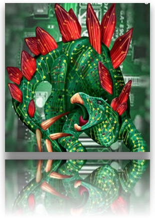
Thread-Level Parallelism

Lecture 5



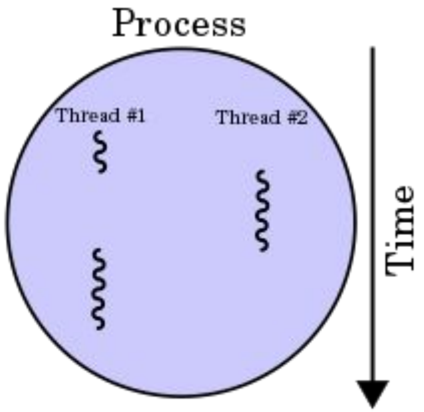
Thread-Level Parallelism

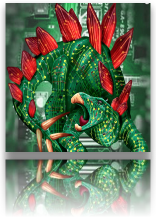
- ★ Threads (Revisit)
- ★ Superscalar/Multiprocessing & Threads
- ★ Fine-grained Multithreading
- ★ Simultaneous Multithreading
  - ○ Thread-Level Parallelism to Instruction-Level Parallelism
- ★ Impact on Hardware
- ★ Software Model
  - ○ OS Threads (POSIX Thread)
  - ○ OpenMP
  - ○ Apple GCD



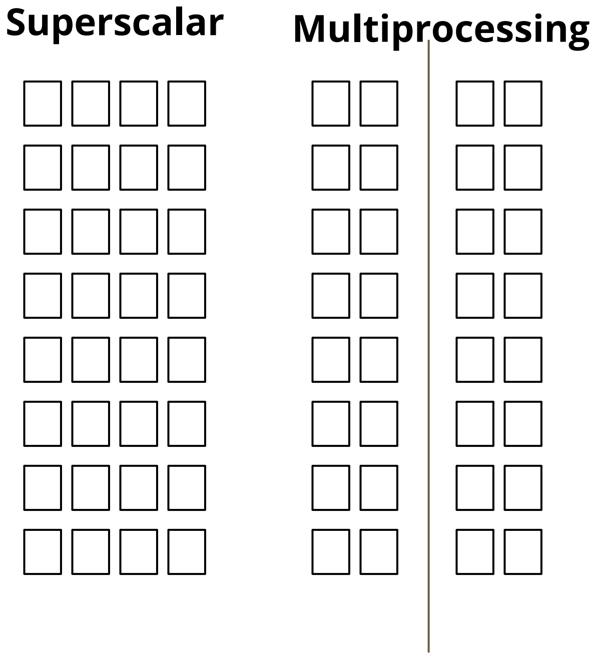
Threads

- ★ Light-weight process
- ★ Multiple threads can exist in a process
    - ○ Shared memory/(global) variables
    - ○ Different PC
    - ○ Different Stack
    - ○ Different registers
- ★ Kernel-level Thread vs. User-level Thread
- ★ Only Kernel-level thread is visible to CPU





Superscalar/Multiprocessing and Threads

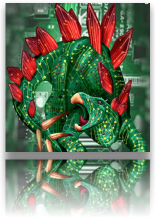


🟥 Thread 1

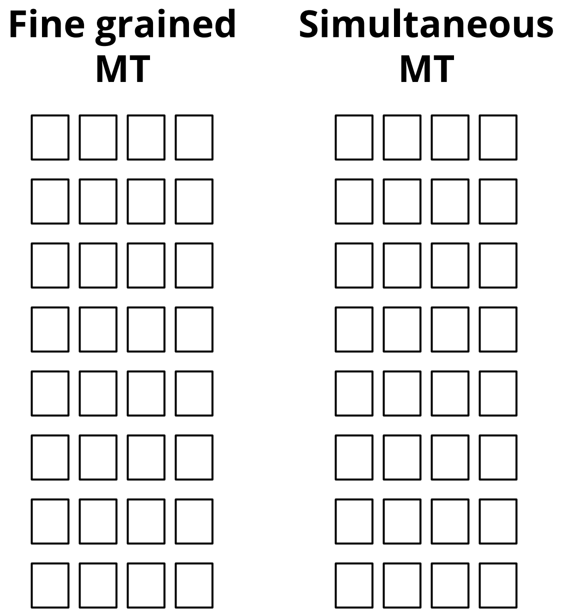
🟦 Thread 2

🟩 Thread 3

- ★ Do we have enough (benchmark) parallelism to fill superscalar and Multiprocessing?
- ★ Can we interchange thread-level parallelism and instruction-level parallelism?



Fine-grained Multithreading & Simultaneous Multithreading

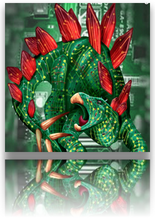


🟥 Thread 1

🟦 Thread 2

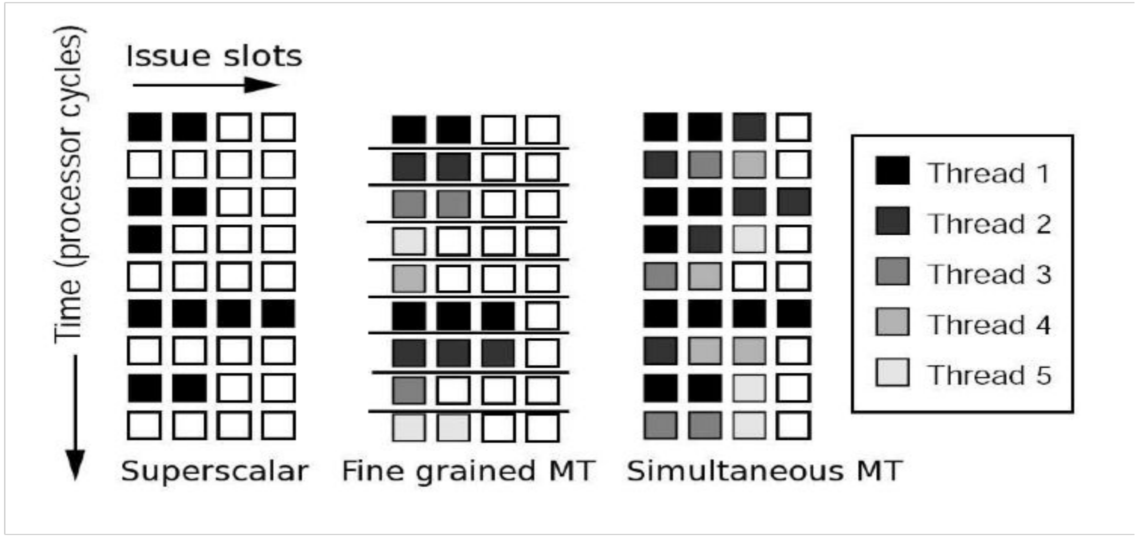
🟩 Thread 3

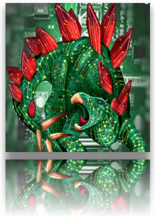
- ★ Fine-grained MT - What if we switch thread every cycle?
- ★ Simultaneous MT - How about we mixed threads in the same cycle?



Simultaneous Multithreading

Comparison





Speedup from SMT

Result from ”Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading,” ACM Transactions on Computer Systems, Vol. 15, No. 3, August 1997, Pages 322–354.

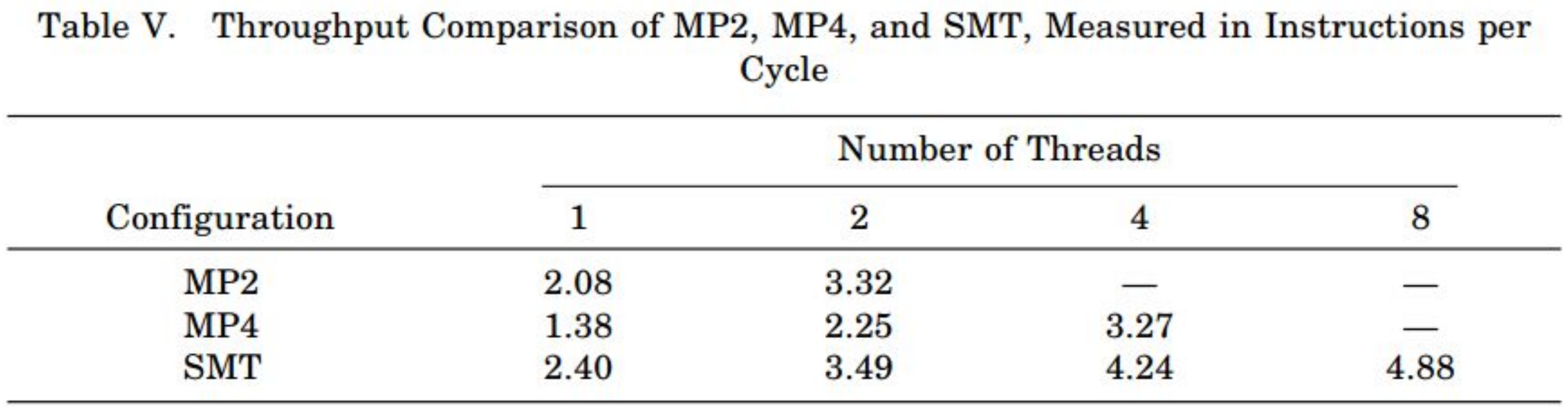
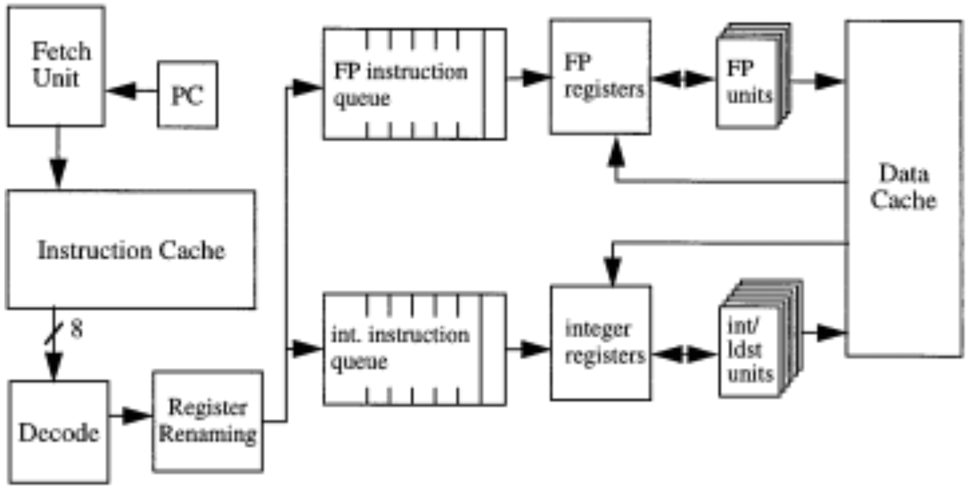


Table V. Throughput Comparison of MP2, MP4, and SMT, Measured in Instructions per Cycle

	Number of Threads			
Configuration	1	2	4	8
MP2	2.08	3.32	—	—
MP4	1.38	2.25	3.27	—
SMT	2.40	3.49	4.24	4.88

Impact on HW



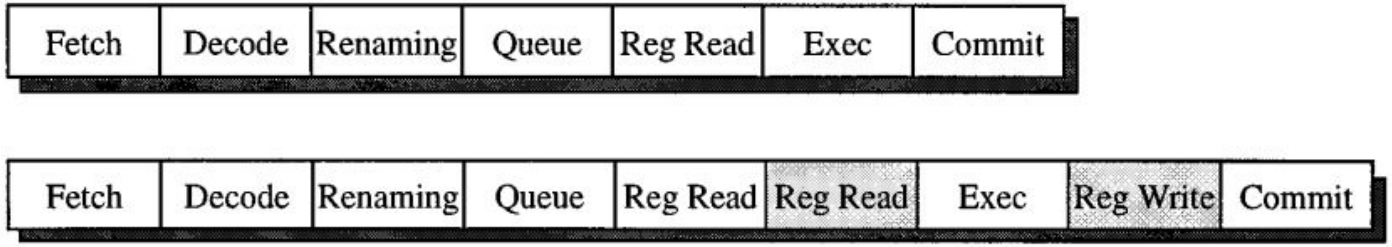
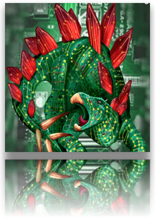
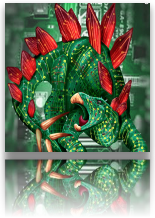


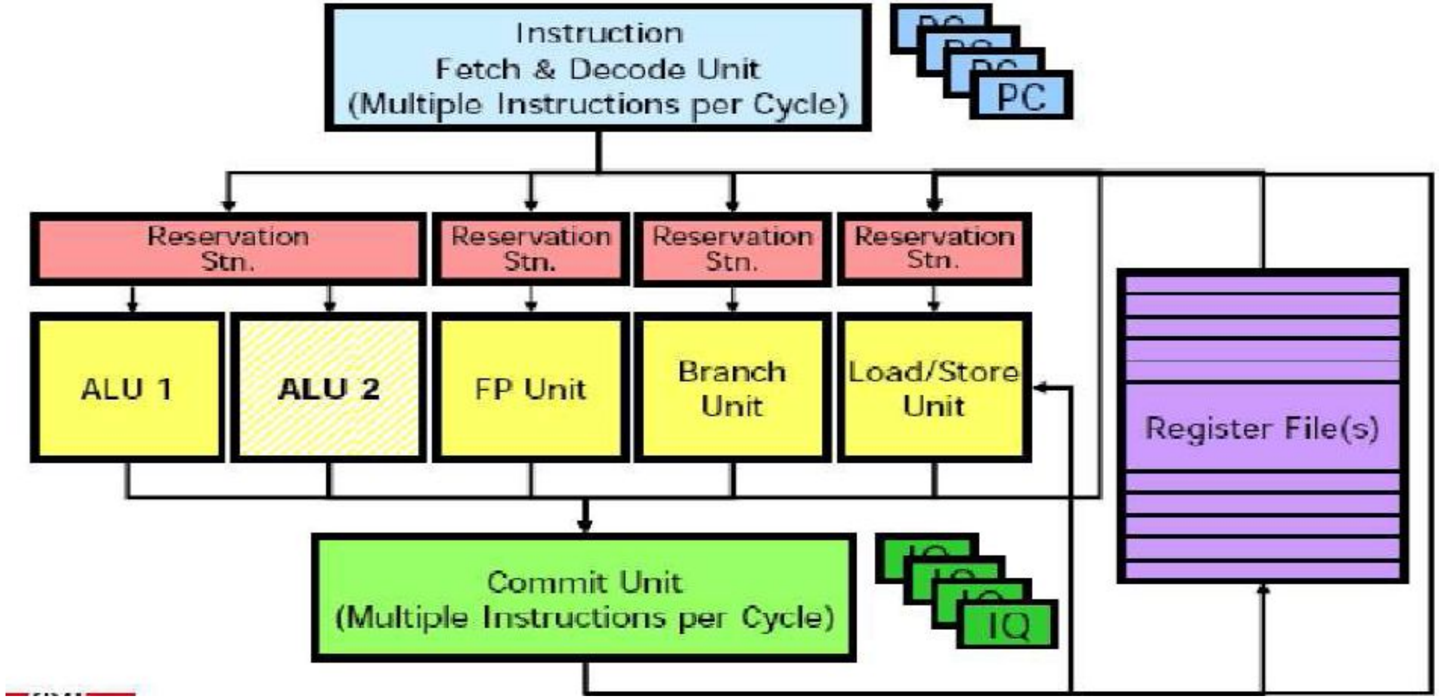
Fig. 3. Comparison of the pipelines for a conventional superscalar processor (top) and the SMT processor's modified pipeline (bottom).



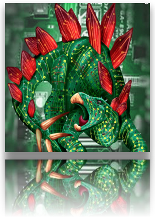
Should we implement it?

- ★ (Minimal) extension to superscalar
- ★ More resources (per thread)
  - ○ Replicated registers, PCs
  - ○ Pipeline flushing/returns
  - ○ Branch predictions?
- ★ Fetch Unit
- ★ Scheduling



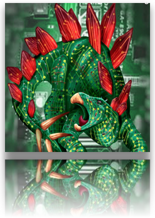


Instruction Fetch & Decode Unit (Multiple Instructions per Cycle)
PC
Reservation Stn.
Reservation Stn.
Reservation Stn.
Reservation Stn.
ALU 1
ALU 2
FP Unit
Branch Unit
Load/Store Unit
Register File(s)
Commit Unit (Multiple Instructions per Cycle)
IQ



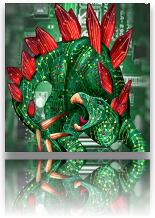
Performance and Implementation Issues

- ★ Performance limitations
  - ○ Issue limits - limit in functional units and availability
  - ○ Too few renaming registers
  - ○ Instruction queue full
- ★ Implementation Issues
  - ○ What to fetch?
  - ○ What to issue?
  - ○ Cache
  - ○ Synchronization
  - ○ Compiler supports



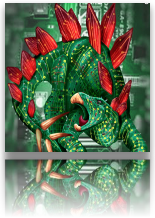
Issue 1: What to fetch?

- static
  - RR
  - 8 instructions from one thread
  - 4 instructions from two threads
  - 2 instructions from four threads
- dynamic
  - Any number of threads with minimal
    - branches?
    - misses?
    - Instructions?



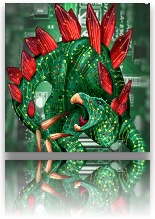
Issue 2: What to issue?

- ★ Aged first?
- ★ Cache-hit speculated last
- ★ Branch speculated first?
- ★ Branches first?



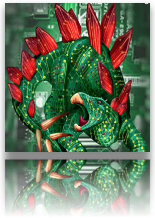
Issue 3: Cache?

- ★ Shared cache?
- ★ Coherence?
- ★ Trashing?



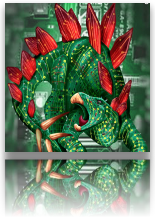
Issue 4: Synchronization?

- ★ lock?
- ★ memory based(atomic instruction)
- ★ acquire (lock)
- ★ release (lock)



Issue 5: Compiler?

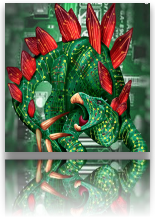
- ★ Minimize cache interference
- ★ Latency
- ★ Sharing optimization



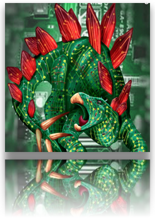
When to use SMT?

★ Biggest application with shared resources
  - e.g. web server

★ What else?



How to multithreading?



Good old POSIX thread (pthread)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 5

void *perform_work(void *arguments){
  int index = *((int *)arguments);
  int sleep_time = 1 + rand() % NUM_THREADS;
  printf("THREAD %d: Started.\n", index);
  printf("THREAD %d: Will be sleeping for %d seconds.\n", index, sleep_time);
  sleep(sleep_time);
  printf("THREAD %d: Ended.\n", index);

}
```

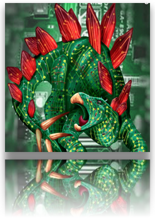
```
int main(void) {
  pthread_t threads[NUM_THREADS];
  int thread_args[NUM_THREADS];
  int i;
  int result_code;

  //create all threads one by one
  for (i = 0; i < NUM_THREADS; i++) {
    printf("IN MAIN: Creating thread %d.\n", i);
    thread_args[i] = i;
    result_code = pthread_create(&threads[i], NULL, perform_work, &thread_args[i]);
    assert(!result_code);
  }

  printf("IN MAIN: All threads are created.\n");

  //wait for each thread to complete
  for (i = 0; i < NUM_THREADS; i++) {
    result_code = pthread_join(threads[i], NULL);
    assert(!result_code);
    printf("IN MAIN: Thread %d has ended.\n", i);
  }

  printf("MAIN program has ended.\n");
  return 0;
}
```



OpenMP

#pragma omp parallel num_threads(10)

★ API for Threads, use directive (#pragma omp …..) to hint compiler

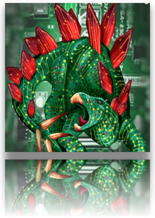
★ Work with C and Fortran

```
// gcc -fopenmp -o loop loop.c
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel for
    for (int i=0;i<100000;i++) {
        a[i]=2*i+i;
        printf("a[%d],%d\n",i, a[i]);
    }
    return 0;
}
```

```
// gcc -fopenmp -o hello hello.c
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```



Grand Central Dispatch

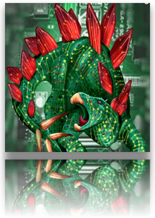
Ported to FreeBSD

- ★ Proposed by Apple
- ★ Used since Mac OS X 10.6 and iOS 4
- ★ An implementation of task parallelism based on the thread pool pattern

```
for (i = 0; i < count; i++) {
      results[i] = do_work(data, i);
}
total = summarize(results, count);
```

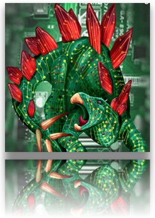
Apache GCD MPM HTTP is 0.3-0.5 LOCs comparing to other MPMs.

```
dispatch_apply(count, dispatch_get_global_queue(0, 0), ^(size_t i){
     results[i] = do_work(data, i);
    });
total = summarize(results, count);
```

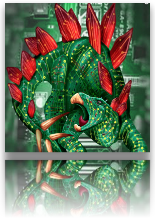


Other tools

- ★ Intel Threading Building Blocks
  - ○ C++ template library developed by Intel for Parallel programming on multi-core processors.
- ★ Java Concurrency
- ★ … etc …



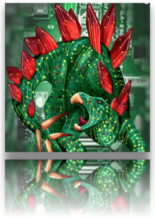
Exercise



Exercises

- ★ What is the main benefit of openmp over standard thread (eg. POSIX thread)?
- ★ For a given code, modify it to take the benefit of simultaneous multithreading processor using openmp. With the new code, what is the potential speed up?

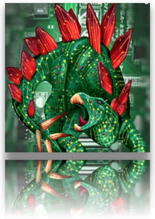
```
#include <stdio.h>
int main(void)
{
    int a[100000];
    for (int i=0;i<100000;i++) {
        a[i]=2*i+i;
        printf("a[%d],%d\n",i, a[i]);
    }
    return 0;
}
```



Exercises

★ Base on OpenMP, explain the concepts of work sharing constructs for
  - loop constructs: for and do
  - sections
  - single
  - Workshare

HINT: https://www.openmp.org//wp-content/uploads/openmp-examples-4.5.0.pdf



End of Lecture 5