Lecture 25: Multi-core Processors

- Today's topics:
  - Writing parallel programs
  - SMT
  - Multi-core examples

- Reminder:
  - Assignment 9 due Tuesday

Shared-Memory Vs. Message-Passing

Shared-memory:

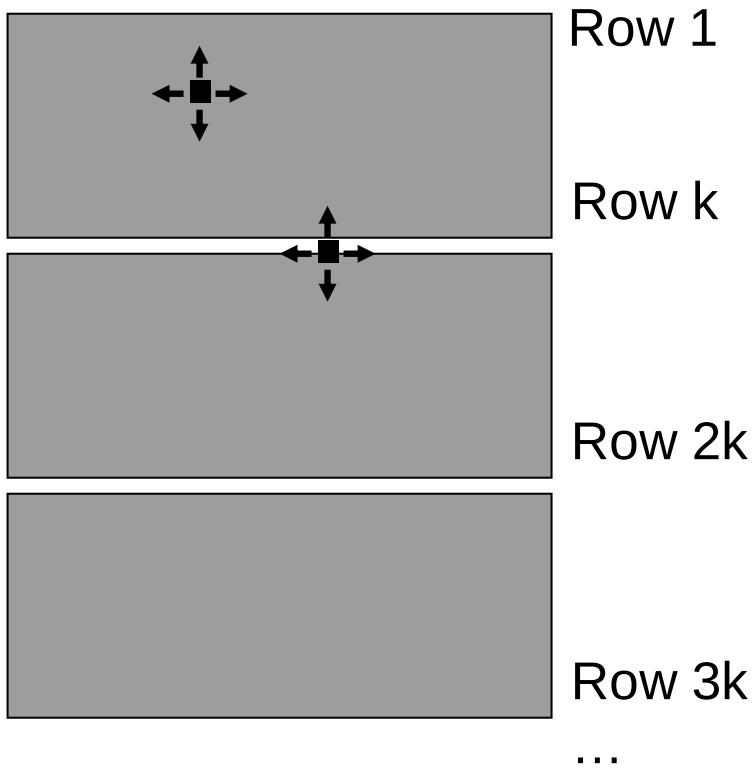
- Well-understood programming model
- Communication is implicit and hardware handles protection
- Hardware-controlled caching

Message-passing:

- No cache coherence → simpler hardware
- Explicit communication → easier for the programmer to restructure code
- Software-controlled caching
- Sender can initiate data transfer

Ocean Kernel

```
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
     diff = 0;
     for i ← 1 to n do
       for j ← 1 to n do
          temp = A[i,j];
          A[i,j] ← 0.2 * (A[i,j] + neighbors);
          diff += abs(A[i,j] – temp);
       end for
     end for
     if (diff < TOL) then done = 1;
  end while
end procedure
```



Shared Address Space Model

```
int  n, nprocs;
float  **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);


main()
begin
   read(n); read(nprocs);
   A ← G_MALLOC();
   initialize (A);
   CREATE (nprocs,Solve,A);
   WAIT_FOR_END (nprocs);
end main
```

```
procedure Solve(A)
    int i, j, pid, done=0;
    float temp, mydiff=0;
    int mymin = 1 + (pid * n/procs);
    int mymax = mymin + n/nprocs -1;
    while (!done) do
      mydiff = diff = 0;
      BARRIER(bar1,nprocs);
      for i ← mymin to mymax
        for j ← 1 to n do
          …
        endfor
      endfor
      LOCK(diff_lock);
      diff += mydiff;
      UNLOCK(diff_lock);
      BARRIER (bar1, nprocs);
      if (diff < TOL) then done = 1;
      BARRIER (bar1, nprocs);
    endwhile
```

Message Passing Model

```
main()
  read(n); read(nprocs);
  CREATE (nprocs-1, Solve);
  Solve();
  WAIT_FOR_END (nprocs-1);

procedure Solve()
  int i, j, pid, nn = n/nprocs, done=0;
  float temp, tempdiff, mydiff = 0;
  myA ← malloc(…)
  initialize(myA);
  while (!done) do
    mydiff = 0;
    if (pid != 0)
      SEND(&myA[1,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      SEND(&myA[nn,0], n, pid+1, ROW);
    if (pid != 0)
      RECEIVE(&myA[0,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
    for i ← 1 to nn do
      for j ← 1 to n do
        …
      endfor
    endfor
    if (pid != 0)
      SEND(mydiff, 1, 0, DIFF);
      RECEIVE(done, 1, 0, DONE);
    else
      for i ← 1 to nprocs-1 do
        RECEIVE(tempdiff, 1, *, DIFF);
        mydiff += tempdiff;
      endfor
      if  (mydiff < TOL)  done = 1;
      for i ← 1 to nprocs-1  do
        SEND(done, 1, I, DONE);
      endfor
    endif
  endwhile
```

Multithreading Within a Processor

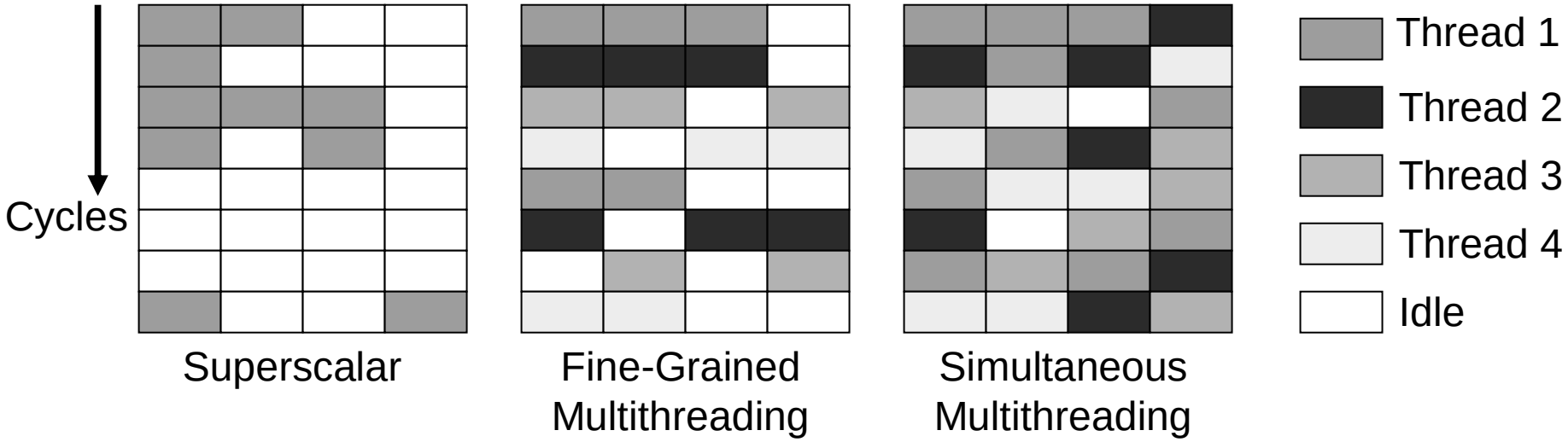
- Until now, we have executed multiple threads of an application on different processors – can multiple threads execute concurrently on the same processor?

- Why is this desireable?
  - inexpensive – one CPU, no external interconnects
  - no remote or coherence misses (more capacity misses)

- Why does this make sense?
  - most processors can't find enough work – peak IPC is 6, average IPC is 1.5!
  - threads can share resources → we can increase threads without a corresponding linear increase in area

How are Resources Shared?

Each box represents an issue slot for a functional unit. Peak thruput is 4 IPC.



- Superscalar processor has high under-utilization – not enough work every cycle, especially when there is a cache miss
- Fine-grained multithreading can only issue instructions from a single thread in a cycle – can not find max work every cycle, but cache misses can be tolerated
- Simultaneous multithreading can issue instructions from any thread every cycle – has the highest probability of finding work for every issue slot

Performance Implications of SMT

- Single thread performance is likely to go down (caches, branch predictors, registers, etc. are shared) – this effect can be mitigated by trying to prioritize one thread

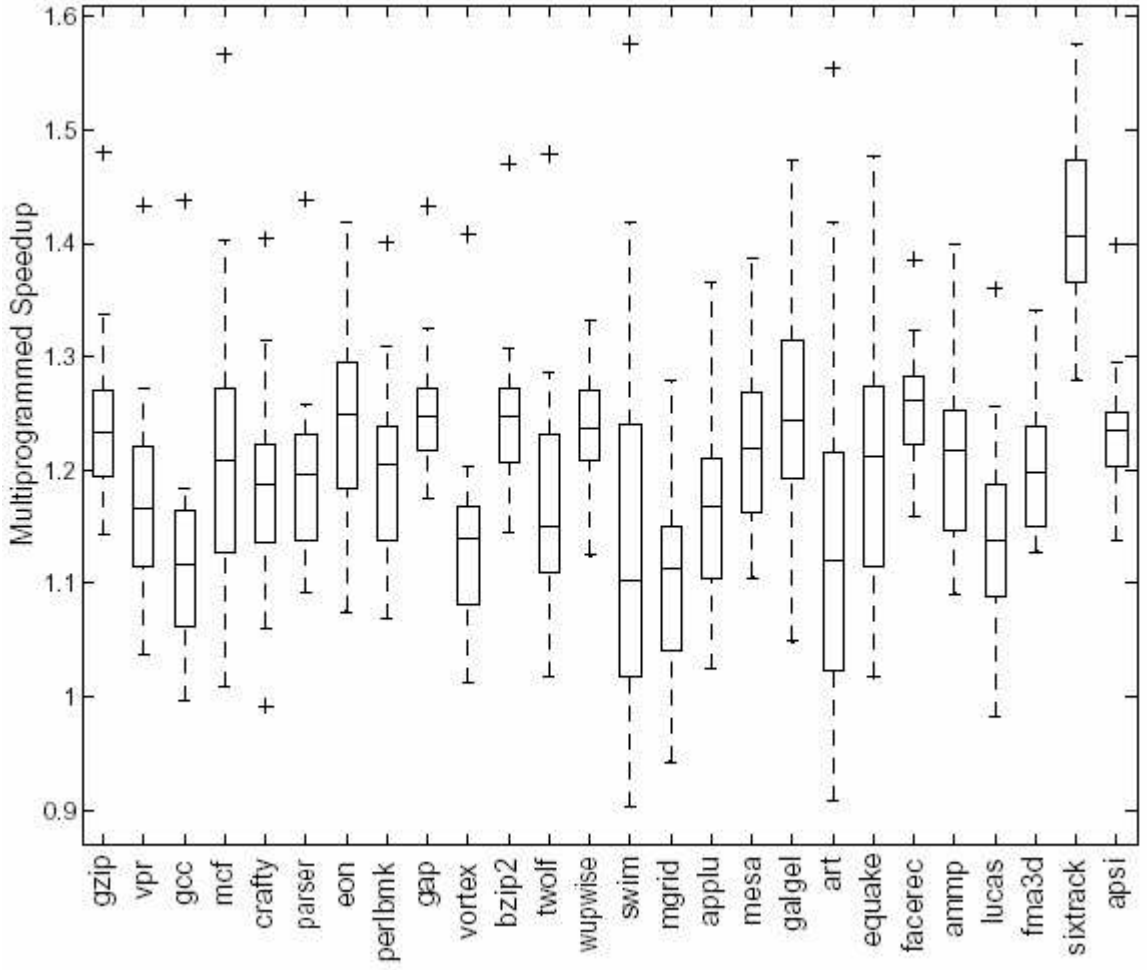
- With eight threads in a processor with many resources, SMT yields throughput improvements of roughly 2-4

Pentium4: Hyper-Threading

- Two threads – the Linux operating system operates as if it is executing on a two-processor system

- When there is only one available thread, it behaves like a regular single-threaded superscalar processor

Multi-Programmed Speedup



Why Multi-Cores?

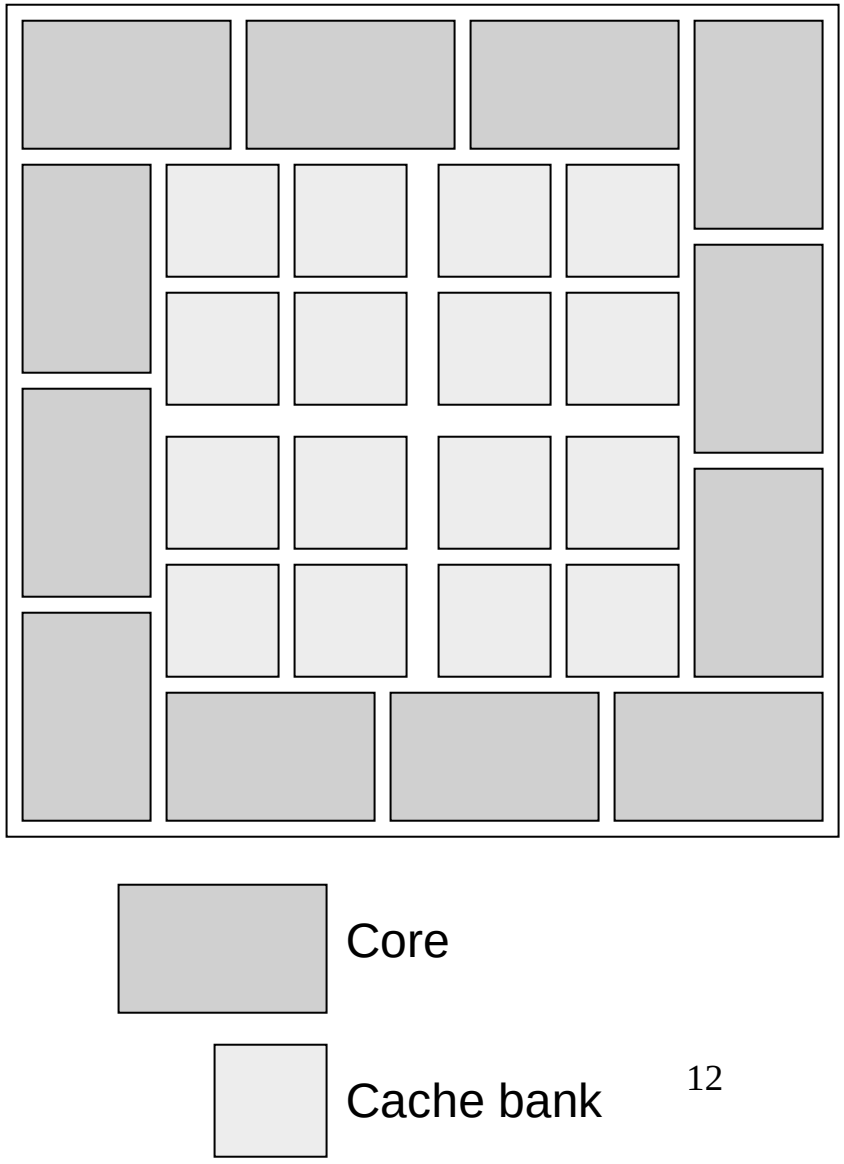
- New constraints: power, temperature, complexity
- Because of the above, we can't introduce complex techniques to improve single-thread performance
- Most of the low-hanging fruit for single-thread performance has been picked
- Hence, additional transistors have the biggest impact on throughput if they are used to execute multiple threads

  … this assumes that most users will run multi-threaded applications

Efficient Use of Transistors

Transistors can be used for:

- Cache hierarchies
- Number of cores
- Multi-threading within a core (SMT)
- Should we simplify cores so we have more available transistors?



Design Space Exploration

Table 3: Maximum AIPC for medium-scale CMTs for SPEC JBB, TPC-C, TPC-W, and XML Test.

Core Config	SPEC JBB 2000				TPC-C				TPC-W				XML Test			
	L1	L2	Cores	AIPC	L1	L2	Cores	AIPC	L1	L2	Cores	AIPC	L1	L2	Cores	AIPC
1p2t	16/32	1.5/12	20	9.8	16/32	2.5/10	16	5.8	16/32	1.5/12	20	8.6	16/32	1.5/12	20	11.8
1p4t	16/32	1.5/12	17	13.2	16/32	2.5/10	14	8.2	16/32	1.5/12	17	10.6	16/32	1.5/12	17	14.8
1p8t	16/32	2.5/10	12	11.7	32/32	1.5/12	14	8.9	32/32	1.5/12	14	13.0	16/32	1.5/12	14	13.8
2p2t	16/32	1.5/12	16	8.6	16/32	1.5/12	16	5.1	16/32	1.5/12	16	7.5	16/32	1.5/12	16	10.5
2p4t	32/32	1.5/12	14	12.9	32/32	2.5/10	12	7.8	32/32	1.5/12	14	10.6	16/32	1.5/12	14	15.2
2p8t	16/32	1.5/12	12	16.5	32/32	2.5/10	9	9.5	32/32	1.5/12	12	13.6	32/32	1.5/12	12	18.9
2p16t	32/64	2.5/10	7	13.3	64/64	2.5/10	7	**11.8**	64/64	1.5/12	9	**15.2**	32/64	1.5/12	9	16.9
3p3t	32/32	1.5/12	13	10.3	32/32	2.5/10	10	5.9	32/32	1.5/12	13	8.5	16/32	1.5/12	13	12.7
3p6t	32/32	1.5/12	11	14.4	32/32	2.5/10	9	8.5	32/32	1.5/12	11	11.3	32/32	1.5/12	11	16.5
3p12t	32/64	1.5/12	9	**17.3**	32/64	2.5/10	7	10.7	64/64	1.5/12	9	14.6	32/64	1.5/12	9	**20.1**
3p24t	32/64	2.5/10	5	13.6	32/64	2.5/10	5	10.9	32/64	1.5/12	6	14.0	32/64	1.5/12	6	15.5
4p8t	32/32	1.5/12	9	14.9	32/32	2.5/10	7	8.5	64/64	1.5/12	9	11.5	16/32	1.5/12	9	16.6
4p16t	32/64	1.5/12	7	16.8	32/64	2.5/10	5	9.8	64/64	1.5/12	7	14.4	32/64	1.5/12	7	18.5
2s1t	64/64	1.5/12	11	4.4	64/64	1.5/12	11	2.8	64/64	1.5/12	11	3.7	64/64	1.5/12	11	5.5
2s2t	64/64	1.5/12	10	7.0	64/64	1.5/12	10	4.3	64/64	1.5/12	10	5.8	64/64	1.5/12	10	8.6
2s4t	64/64	1.5/12	9	10.5	64/64	1.5/12	9	6.4	64/64	1.5/12	9	8.7	64/64	1.5/12	9	12.4
2s8t	64/64	1.5/12	7	12.1	64/64	1.5/12	7	8.1	64/64	1.5/12	7	10.6	64/64	1.5/12	7	12.7
4s1t	64/64	1.5/12	7	2.9	64/64	1.5/12	7	1.9	64/64	1.5/12	7	2.6	64/64	1.5/12	7	3.7
4s2t	64/64	1.5/12	6	4.5	64/64	1.5/12	6	2.9	64/64	1.5/12	6	3.9	64/64	1.5/12	6	5.8
4s4t	64/64	1.5/12	5	6.6	64/64	1.5/12	5	4.1	64/64	1.5/12	5					
4s8t	64/64	1.5/12	4	8.5	64/64	1.5/12	4	5.5	64/64	1.5/12	4					

Note: The L1 refers to the primary data/instruction cache size. The L2 cache configuration size (MB
with the total number of cores for that CMT configuration.

From Davis et al., PACT 2005

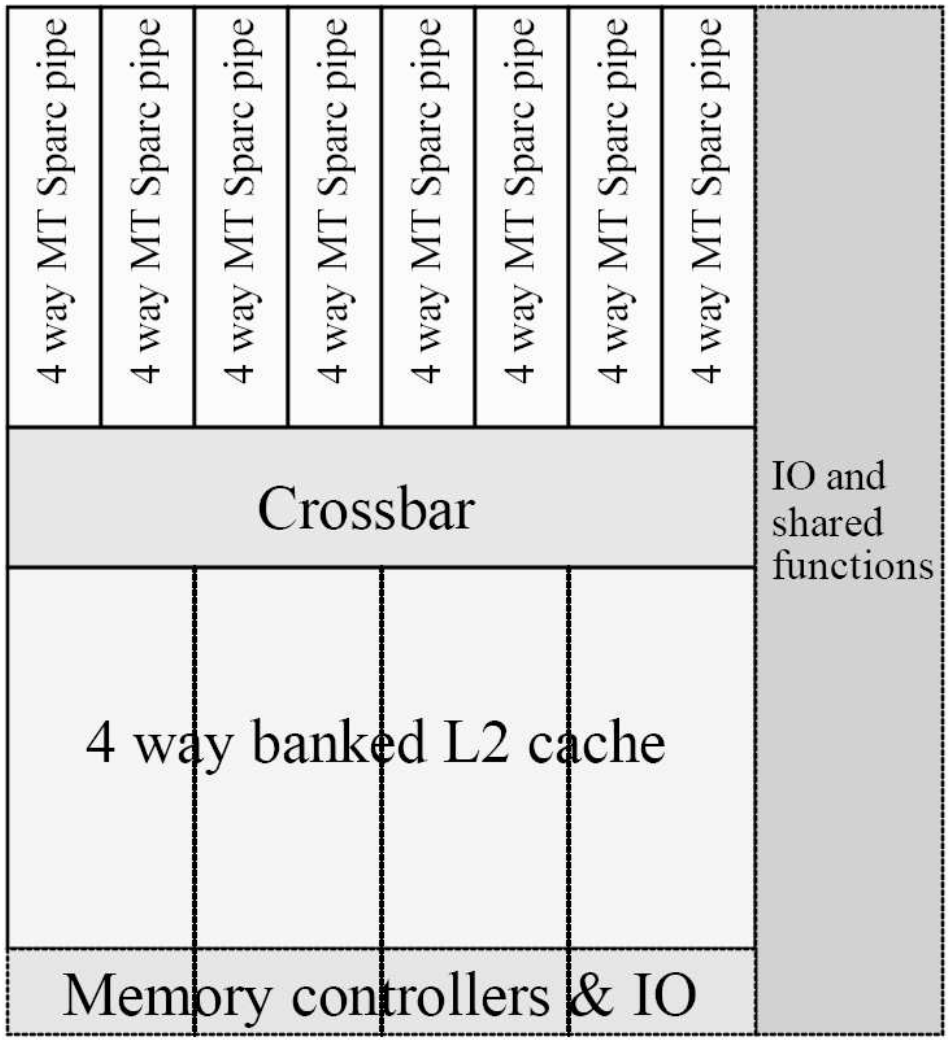
p – scalar pipelines
t – threads
s – superscalar pipelines

Case Study I: Sun's Niagara

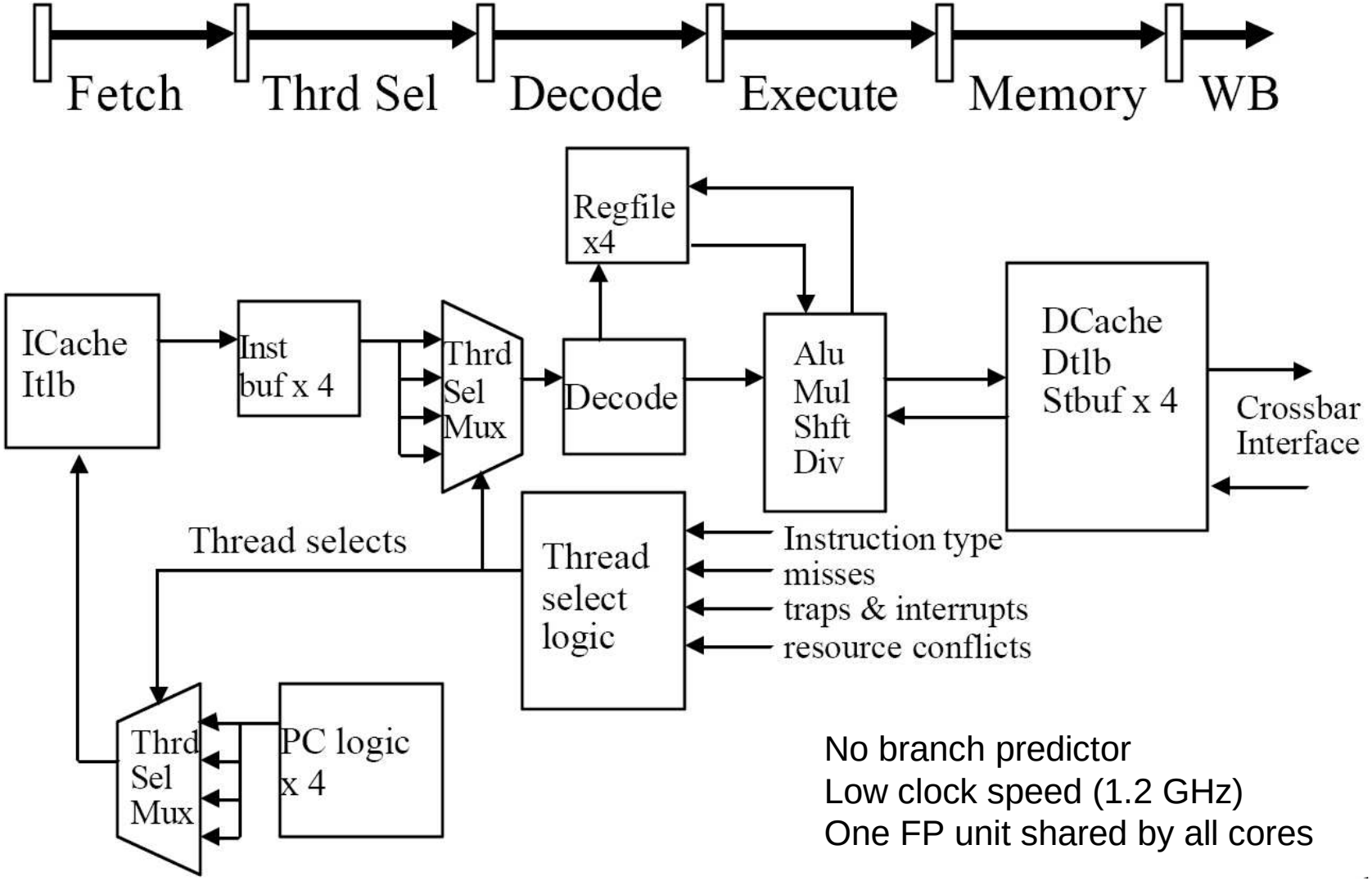
- Commercial servers require high thread-level throughput and suffer from cache misses

- Sun's Niagara focuses on:
  - simple cores (low power, design complexity, can accommodate more cores)
  - fine-grain multi-threading (to tolerate long memory latencies)

Niagara Overview



SPARC Pipe



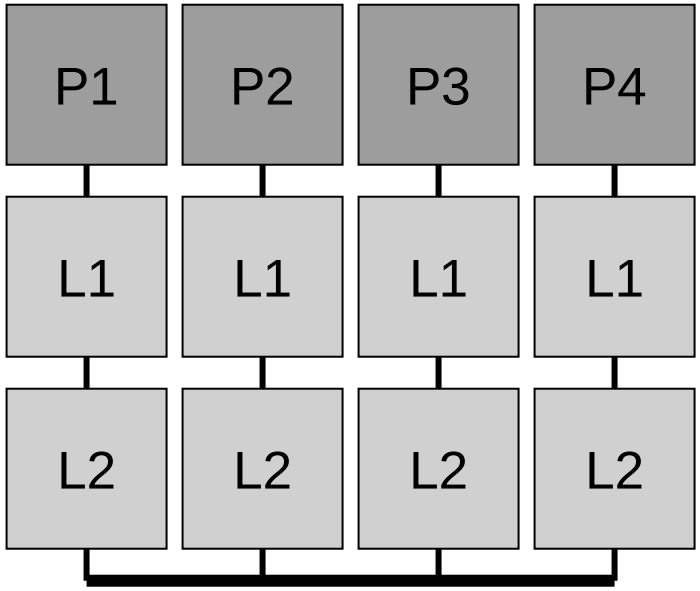
No branch predictor
Low clock speed (1.2 GHz)
One FP unit shared by all cores

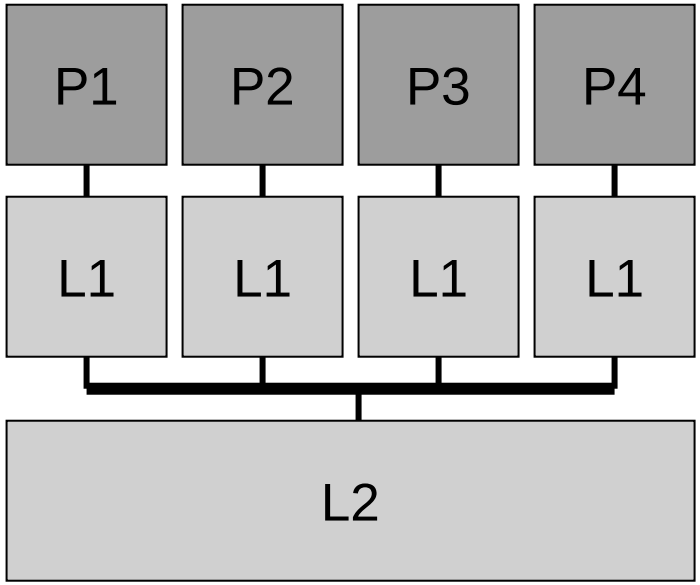
Case Study II: Intel Core Architecture

- Single-thread execution is still considered important ➔
  - out-of-order execution and speculation very much alive
  - initial processors will have few heavy-weight cores
- To reduce power consumption, the Core architecture (14 pipeline stages) is closer to the Pentium M (12 stages) than the P4 (30 stages)
- Many transistors invested in a large branch predictor to reduce wasted work (power)
- Similarly, SMT is also not guaranteed for all incarnations of the Core architecture (SMT makes a hotspot hotter)

Cache Organizations for Multi-cores

- L1 caches are always private to a core
- L2 caches can be private or shared – which is better?





Cache Organizations for Multi-cores

- L1 caches are always private to a core
- L2 caches can be private or shared
- Advantages of a shared L2 cache:
  - efficient dynamic allocation of space to each core
  - data shared by multiple cores is not replicated
  - every block has a fixed “home” – hence, easy to find the latest copy
- Advantages of a private L2 cache:
  - quick access to private L2 – good for small working sets
  - private bus to private L2 → less contention

Title

- Bullet