Presentation 7

Beyond Instruction Level Parallelism

בעשור האחרון שיטת superscalar הגיעה לגבולות התחום שלה והיכולת לשפר ביצועים. באותן השנים השתפרו טכנולוגיות שמאפשרות ייצור מספר רב של ליבות בשבב סיליקון אחד, וניצול הליבות על ידי חלוקת תוכנה לפתילים (threads). מצד אחד, הטכנולוגיות האלו מעודדות מעבר מדגש על מקבילות ברמת הפקודות (ILP) לדגש על מקבילות ברמת הפתיל (TLP). ומצד שני, תכנות מקבילית היא לא תחום חדש ועדיין סובלת מבעיות תיאורטיות שלא גילו להן פתרון בחמישים השנים שעוסקים בה. הפרק הזה עוסק בשיטות כלליות בתכנות מקבילית. טכנולוגיות פרטניות לארכיטקטורות מקבילות יוצגו בפרקים 8 ו- 9.

Slide 2 – 3 Summary of Superscalar Processing

Slide 2 shows a general pipeline for a superscalar CPU core with dynamic scheduling, and summarizes the techniques introduced for recent generations of RISC processors to improve performance:

In the IF stage, branch prediction and trace cache minimize branch penalties.

In the IF stage, prefetch minimizes cache miss penalties.

In the ID stage, virtual registers and automatic register renaming prevent false data dependencies and permit greater ILP.

In the ROB, out-of-order execution and in-order retirement allow instructions to be executed in the most efficient order while maintaining atomic semantics at the level of machine code.

Multiple execution units permit multiple instructions to be executed in every clock cycle.

A stream buffer minimizes cache miss penalties in the data cache access, and a memory reorder buffer permits efficient scheduling of loads and stores.

Slide 3 reviews the difficulty in scaling these superscalar methods.

Slide 4 – 7 Sequential and Parallel Operations

The state of the art in programming and hardware implementation meet in an awkward manner at the level of machine language. Most high-level programs combine **parallel and sequential constructs**, using model-dependent sections: processes, threads, classes, procedures, and control blocks. But these sections are compiled for a particular ISA made up of low-level CPU operations: data transfers, arithmetic/logic operations, and branch operations. When the high-level job is executed, the CPU executes machine instructions, small sequential operations with only local information about 2 or 3 operands. The CPU cannot recognize the abstract model-dependent structures used by the programmer — the information about inherent parallelism is lost in translation to the CPU. The sophisticated hardware techniques described on slide 2 can be seen as attempts to rediscover some of the parallelism intended by the programmer, using heuristics such as prediction, renaming, cache, and inherent parallelism.

Concurrent tasks are defined to run independently, whether in parallel (executing simultaneously on multiple copies of hardware) or interleaved (a single hardware unit alternates between tasks). For example, a program may respond to mouse events, respond to keyboard input, and accept a network message concurrently.

Concurrency can be divided into two types:

In **functional concurrency**, each task performs a different set of operations. For example, the rotation of a vector in the plane maps the vector A to A' = R(θ) × A where R is a rotation matrix. As a mathematical abstraction, this rotation takes place concurrently, but the sequential implementation in code performs two slightly different operations

$$A_x' = A_x \cos\theta + A_y \sin\theta$$

$$A_y' = -A_x \sin\theta + A_y \cos\theta$$

In **data concurrency**, each task performs the same operation on different data. For example, the sum of two vectors is performed concurrently as the mathematical abstraction C = A + B, but the implementation in sequential code performs the operations

```
for (i = 0, i < n, i++) C[i] = A[i] + B[i]
```

While high-level languages permit the definition of concurrent structures such as vectors and matrices, compilation for a sequential machine language loses most of the information about concurrency. The hardware applies heuristics in order to recover some of this lost information.

In **parallel programming**, the programmer identifies inherently parallel operations in the high-level job, designs a **parallel algorithm** and translates the job into source code that specifies parallel operations to the compiler. Parallel threads for are defined for functional and data decomposition. The OS receives deterministic instructions reflecting the inherent parallelism, including code and threading instructions. The hardware disperses instructions defining as vectorized operations, pre-grouped independent operations, and thread level parallelism, to multiple processors or execution units.

Slide 8 – 9 The "Old" Parallel Processing

Early R&D in the field of parallel processing experienced growth and decline:

1958 Research was begun at IBM on parallelism in arithmetic operations

1960 – 1980 Mainframe computers were designed as symmetric multiprocessors (SMP) with 4 to 24 CPUs. The OS dispatches processes from a shared ready process queue to idle processors.

1980 – 1995 A research boom led to a deeper understanding of parallel processing, but limited practical success:

Automated parallelization by the compiler achieved limited success. Compilers cannot easily identify inherent parallelism.

Parallel constructs in high-level languages require a long learning curve. Parallel programmers are typically specialists.

Inherent complexities were identified:

Processing and communication overhead includes inter-process message passing (to spawn and assemble tasks with many CPUs) and synchronization to prevent race conditions (data hazards).

Data structures for the shared memory model must be defined with good blocking to cache organization, to avoid a cache bottleneck.

1999 It was fashionable to consider parallel processing a dead end that would never provide good performance in a reasonable code development cycle.

The graph on slide 9 shows the relative number of papers on multiprocessors (in red) and their interconnection networks (in green) submitted to the International Symposium on Computer Architecture. As seen in the graph R&D reached a peak around 1980 to 1995, and dropped off sharply after 1996.

Slide 10 It's Back — the "New" Parallel Processing

The new push toward multicore processing was prompted by the limits in earlier strategies for improving CPU performance. As seen in slide 3, superscalar growth has reached practical limits.

Although clock speed tripled from 1 GHz to 4 GHz between 2000 and 2005, faster speeds face physical limits. The time required for an electrical signal to cross a 3 cm of empty space at the speed of light is one clock cycle of a 10 GHz clock, making time synchronization difficult. Another problem is removal of heat from the CPU core. Digital circuits conduct electrical current during 0 to 1 and 1 to 0 transitions, so faster clock rates mean higher power consumption. As chips get larger, it becomes more difficult to remove the heat from the center of the CPU, causing damage to the circuits.

The "old" parallel processing technology, based on assigning tasks to processors in physically separate chips is still not sufficient. The new developments that make multicore CPUs viable include the low price of placing several cores in the same physical package, compiler support for program decomposition into lightweight threads along with debugging tools, and OS level thread scheduling.

Slide 11 – 13 Processes and Threads

A **process** (task) is one instance of an independently executable program. It was the basic unit of OS kernel scheduling on a traditional non-threading kernel. A process is associated with an entry in a process control block (PCB) that defines resources: ID, state, PC, register values, stack and memory space, I/O descriptors, among others. A task (context) switch is a high volume transfer operation. A process may be organized into one or more threads owned by the process.

A **thread** is one instance of an independently executable instruction sequence that is not organized into smaller multitasked units. Threads own limited private resources — a PC, a stack, and its registers values. Other resources are shared with other threads owned by the parent process. Threads may be scheduled by the OS kernel or by user code. A thread switch is a low volume transfer operation.

One standard example of a multithreaded program is a web server. The server listens on an open socket and creates a new socket for each accepted client connection. Since client connections are independent the server can create a new thread to handle the new client.

The multithreaded web server is an example of **functional decomposition**. In a 3D game, functional decomposition could be the creation of independent threads for handling the ground, the sky, and a character. Higher performance can often be achieved with **data decomposition**, such as the division of the sky into n sections and the creation of an independent thread to update each section.

Slide 14 – 17 Hardware Implementation of Multithreading

Multithreading does not impose any special hardware requirements. Multithreaded code can run on a single core or multicore processor. The run-time efficiency depends on the interaction between hardware and software. The five basic approaches to multithreading are:

Coarse-grained multithreading	A single CPU switches to a new thread on a long stall
Fine-grained multithreading	A single CPU switches to a new thread on each clock cycle
Simultaneous multithreading (SMT)	A superscalar pools instructions from multiple threads in the ROB, creating a large instruction window
Hyper-Threading	Intel technology combining fine-grained multithreading and SMT
Multiprocessing	A scheduler dispatches threads to multiple independent cores

The upper illustration on slide 15 shows the utilization of EUs on a **single threaded superscalar** CPU over time. In CC1 (at left), two EUs execute instructions in parallel while two EUs are idle. In CC2, three EUs work in parallel while one EU is idle. In CC3, only one EU performs useful work. IN CC7 and CC8, the program is stalled and no EU is actively executing an instruction.

The center illustration shows the utilization of EUs on a **coarse-grained multithreading** CPU over time. On CC1 to CC4, thread 1 (blue) runs, utilizing 2, 3, 1, and 4 EUs. On CC5 to CC7, thread 2 (green) runs, utilizing 2, 4, and 3 EUs. On CC 8 to CC10, thread 3 (yellow) executes, and on CC11 and CC12, thread 4 (pink) executes.

The bottom illustration shows the utilization of EUs on a **fine-grained multithreading** CPU over time. On CC1 thread 1 (blue) runs, on CC2, thread 2 (green) runs, On CC 2, thread 3 (yellow) runs, and so on.

Notice that course-graining and fine-graining can usually insure that the pipeline is kept busy on a stall, but cannot insure high EU utilization.

Slide 16 shows the utilization of EUs on a **simultaneous multithreading** CPU over time. The ROB holds many instructions from many independent threads. Because the threads are independent, there are no data dependencies between instructions of different threads, creating a large instruction window. When a thread is cancelled following a misprediction, only the instructions associated with that thread are deleted from the ROB — about 75% of the instructions in the ROB remain valid, eliminating much of the decode bottleneck (slide 2 in presentation 5).

Slide 17 shows Intel Hyper-Threading technology (see slide 55 in presentation 3).

Slide 18 – 20 Thread Coexistence

Multithreaded code provides a source of independent instructions, permitting high processor utilization on a single SMT superscalar core. Although independent applications running in parallel should produce no data dependencies, their independence can actually create memory resource conflicts. For multiple threads running, each application requires different data blocks in to be in cache, and each creates a different branch prediction cache and trace cache. Also, parallel threads of a single application run in a coordinated fashion, communicating, synchronizing, and exchanging data. A stall in one thread can stall related threads, through cache misses, page faults, branch mispredictions, and so on.

The helper thread model is intended to coordinate memory conflicts. Helper threads perform no useful work or change any program result — their results not committed to memory. They perform loads and branches that appear in the work of other threads. The CPU encounters cache misses in the helper thread and updates cache, before the working thread requires the data. As an example, the code at the top of slide 20 suffers a cache miss on the load and a branch misprediction on the branch at the end of the loop. But pairing this code with a helper thread, that merely performs a load from the same data address and branches on the same instruction, preloads the data block to cache and prepares the branch predictor to recognize the taken branch.

Slide 21 Flynn Taxonomy for CPU Architectures

See slide 4 in presentation 3 for a discussion of the Flynn Taxonomy.

Slide 22 – 25 Multiprocessor Architecture

The top of slide shows the structure of a typical **SISD / SIMD workstation**. The central feature is the bus adapter chipset (usually PCI). The adapter is connected at left to a dual core CPU, connected at right to main memory, and connected below to the I/O bus that connects peripheral hardware (storage, user interface, network connections, and so on).

A general **MIMD multiprocessor** contains multiple CPUs, each with multiple cores. The cores are connected to main memory (localized or distributed across the system) through some interconnection network (also called a fabric). The network may be as simple as a data bus (such as PCI) or a specialized switching fabric. The network also connects the CPUs and memory to the I/O system. Multiprocessors are usually characterized by the relationship of CPUs to memory, a relationship that also determines the software parallelization model.

A **shared memory system** is shown at the top of slide 23 (discussed at length in presentation 8). In shared memory, the global memory space A may be physically partitioned into M blocks (0 to M – 1) but the full memory appears unified through the switching fabric. Each of the N processors (0 to N – 1) can access the full address space (address 0 to A – 1) via the internal network. Threads running on the N CPUs communicate among themselves by writing and reading shared addresses in memory. For many activities, shared memory requires synchronization of memory accesses to prevent data hazards.

A **message passing system** is shown at the bottom of slide 23 (discussed at length in presentation 9). A node consists of a CPU with its private address space A (0 to A –1). The multiprocessor is made up of N nodes (0 to N – 1). CPUs cannot generally access the memory of any other node. Processors communicate by formulating, passing, and reading inter-CPU messages over the internal network. The messages combine data and memory synchronization information.

Slide 24 shows the Flynn-Johnson Taxonomy, a revision that distinguishes shared memory from message passing, and global memory (located in a centralized location) from distributed memory. In this framework, most shared memory machines are either GMSM or DMMP.

The characteristics of the shared memory model and the message passing model are compared in slide 25. In shared memory systems, the synchronization of memory accesses, implemented by maintaining the coherence of cache in the multiple CPUs is a source of communication overhead. In message passing, cache coherency is not required, but the passing of messages contributes a similar overhead. A major difference between models is scalability. Shared

memory systems are limited in scale by the exponential complexity of shared access by multiple CPUs to the memory system. The sharing also requires the CPUs to be physically close, and is applicable to fine grain parallelism — light, parallel threads with short code length and small data volume. Message passing is limited only by network capacity (the CPUs may be separated by thousands of miles) and is applicable to course grain parallelism — heavy parallel threads with long code length and large data volume. Shared memory programming is facilitated by pthreads and the OpenMP model that defines parallel constructs for C/C++. Message passing programming is facilitated by the Message Passing Interface (MPI) API.

Slide 26 – 30 Amdahl's Law for Multiprocessors

Amdahl's equation for parallel processing was obtained on slide 25 of presentation 1. It is derived again on slide 26 by defining IC_P (the number of instructions that can be run in parallel) as $F_P \times IC$, where F_P is the fraction of the program that can be run in parallel. The remaining part of the program, $1 – F_P$, runs sequentially.

Slide 27 shows results obtained by Intel for the speedup of multiprocessor systems operating on programs written to handle **on line transaction processing** (OLTP). Without Hyper-Threading (which contributes a 20% improvement as discussed in slide 56 of presentation 3), the speedup is found to be 1.7 for N = 2 CPUs and 2.6 for N = 4 CPUs. Inverting Amdahl's equation to find F_P one finds that about 80% of the instructions can be run in parallel. Slide 26 shows that with this proportion, the maximum speedup as $N \rightarrow \infty$ is S = 5, corresponding to an ideal CPI = 0.20.

Some additional details on OLTP are given on slide 28. The transactions processed in OLTP are client requests to a server accessing a database, for example, banking requests, order processing, inventory management, accesses to the student information system, and so on. The requests are inherently independent and so the workload in the server is inherently multithreaded, with one thread per request. The server sees a large batch of small parallel threads. The code is generally short and inherently sequential — SQL transactions made up of accesses to multiple tables. The multi-table accesses produce a severe memory latency of several CPU stalls per thread. Running the SPEC Cint benchmark suite on a certain 8-pipeline dynamic scheduling superscalar led to a measured CPI = 0.31 (equivalent to $CPI^{stall} = 1.48$, counting idle EUs as stall cycles). Running an OLTP load on the same hardware leads to a measured CPI = 1.27 (equivalent to $CPI^{stall} = 9.16$). General purpose, server-oriented CPUs (such as Intel's Xeon family) do not run OLTP efficiently. For this reason, Oracle (whose database software specializes in OLTP) purchased Sun Microsystems (once the leader in high performance workstations) to obtain the SPARC CPU, better suited to simultaneous multithreading (SMT) than general purpose dynamic scheduling.

A measure of multiprocessor efficiency is found from $E = S / S_{ideal}$ where S is actual speedup and S_{ideal} is the speedup for $F_P = 1$ (full parallelization). It is shown on slide 35, that unless F_P is exactly 1, the efficiency goes to zero for a large number of processors. This simply means that throwing a lot of processors at a problem with a significant sequential component is not efficient.

Slide 31 – 32 Grosch's Law versus Amdahl's Law

Long before Amdahl's work, Herbert Grosch at IBM studied computer performance and found that the performance-to-price ratio improves linearly with price (more performance per dollar at higher prices). This is formalized as

$$\text{Cost}(N) = \alpha \times N \quad \Rightarrow \quad N = \text{Cost}(N)/\alpha$$

$$\text{performance}(N) = k_G \times (\alpha \times N)^2 = k_G \alpha^2 N^2 \quad \Rightarrow \quad \frac{\text{performance}(N)}{\text{cost}(N)} = k_G \alpha^2 N$$

$$S = \frac{\text{performance}(N)}{\text{performance}(1)} = N^2$$

But Amdahl's law implies that

$$\text{performance}(N) = \frac{k_{Amdahl}}{(1-F_P) + \frac{F_P}{N}} = \frac{k_{Amdahl}}{(1-F_P) + \frac{F_P}{\text{Cost}(N)/\alpha}} = \frac{\text{Cost}(N)}{(1-F_P)\text{Cost}(N) + \frac{\alpha}{k_{Amdahl}} F_P}$$

$$\frac{\text{performance}}{\text{Cost}(N)} = \frac{1}{(1-F_P)\text{Cost}(N) + \frac{\alpha}{k_{Amdahl}} F_P} \quad \text{for some constant } k_{Amdahl}$$

So, by Amdahl's law the performance-to-price ratio should decline with rising cost (diminishing returns from throwing a lot of processors at a problem that has a significant sequential component).

The difference between Amdahl's and Grosch's results can be understood by looking at two different types of computability problem.

On the one hand, a typical problem running in a shared memory environment will initialize a few variables, execute a loop over a shared data set of MB scale, and display the result. The initialization and display will always execute sequentially, even if every loop iteration is handled in parallel on a separate core. No additional improvement can be expected by making the CPU number larger than the number of loop iterations.

On the other hand, a typical problem running in a message passing environment will initialize a few variables, execute a long loop over a data set of GB to PB scale (1 PB = 1Mega GB), and display the result. Such problems include large simulations, weather forecasting, and analysis of astronomical data. Because these data sets can usually be decomposed into effectively independent sets (with periodic exchange of data at the boundaries between sets), message passing problems typically scale with the size of the problem. The Gustafson-Barsis Law applies to large problems in which the number of processors grows with the size of the problem. The run time is expressed as a sequential part and a part that grows with problem size.

$$\text{run time in serial execution} = s + p \times n \text{ , } n = \text{size of problem}$$

$$\text{speedup compared to serial execution} = \frac{s + p \times n}{s + p} \xrightarrow[n \text{ large}]{} n$$

As a proportion of instructions to run, the sequential part becomes very small as the problem (and number of CPUs) grows. The Gustafson-Barsis Law is equivalent to the Grosch result taking s = 1 instead of s = 2.

Slide 33 Interconnection Network Types

Slide 33 shows a taxonomy of **interconnection networks** for multiprocessor systems. The basic distinction is between static (permanent) connections between nodes and dynamic connections established on demand. Static connections can provide either full node-to-node connectivity or limited connectivity that requires multi-hop message transport.

Dynamic interconnection networks are divided into shared medium transport, typically a data bus, over which nodes must arbitrate (compete) for access to the network, and switched transport, in which the switching elements are specially configured for each message.

Slide 34 – 35 Communication Overhead and Amdahl's Law

Communication, whether for shared memory or message passing, creates parallelization overhead that affects performance. In analogy with instruction processing time, the time required for communication can be written as

$$T^{comm} = CPI^{comm} \times IC_p \times \tau = CPI^{comm} \times F_p \times IC \times \tau$$

CPI^{comm} = processor clock cycles devoted to communication per instruction executed in parallel

F_p = fraction of program running in parallel

Defining

$$F_{overhead} = \text{overhead factor} = CPI^{comm} / CPI,$$

which characterizes the ratio of communications activity to computing activity, the speedup compared to 1 CPU becomes

$$S = \frac{CPI \times IC}{CPI \times IC \times (1 - F_p) + \frac{CPI}{N} \times F_p \times IC + CPI^{comm} \times F_p \times IC} = \frac{CPI}{CPI \times (1 - F_p) + \frac{CPI}{N} \times F_p + CPI^{comm} \times F_p}$$

$$= \frac{1}{(1 - F_p) + F_p \left(\frac{1}{N} + F_{overhead} \right)},$$

a modified form of Amdahl's equation. For a large number of processors,

$$S = \frac{1}{1 - F_p (1 - F_{overhead})}.$$

When the overhead factor approaches 1, meaning that communication activity is comparable to computing activity, the speedup drops to 1, even for a large number of CPUs.