Chapter 9

Advanced Architectures

הטכנולוגיות שהוצגו בפרקים 1 עד 8 מציירים תמונה של ה- state of the art עד 1995 בערך. כבר אז מעבד RISC בתוספת branch prediction היה מסוגל לבצע פקודות integer ב- CPI קרוב ל- 1. אין צורך לומר שפיתוח מעבדים לא נגמר לפני 20 שנה. בעיקר המאמץ מאז התרכז בשיפור ביצוע פקודות FP ושיפור ה- CPI לפקודות integer לערכים קטנים מ- 1 (סיום יותר מפקודה אחת בכל מחזור שעון).

Slides 2 – 5 General Outlook

The basic parameters of **performance** are $T = CPI \times IC \times \tau$ where $CPI = CPI^{ideal} + CPI^{stall}$. To improve run time requires reducing CPI, IC, or τ. Because IC grows with software complexity there is little likelihood of making programs smaller. This is especially true because software complexity grows as new methods are introduced to decrease software development time. Also the clock speed cannot be raised significantly because of the fundamental physical limitations seen on slide 34 in chapter 1.

This points to CPI, with 4 possible areas of improvement:

Instruction and thread level parallelism to achieve $CPI^{ideal} < 1$

Reducing instruction dependency stalls to lower $CPI^{stall}_{data\ dependency}$

Reducing cache latency to lower $CPI^{stall}_{cache\ miss}$

Reducing branch stalls to lower $CPI^{stall}_{branch\ penalty}$

There has been some success in all of these areas:

Instruction Level Parallelism (ILP) is improved by providing multiple copies of hardware units in each processor. Multiple instructions begin executing on the same clock cycle. Several instructions finish on every clock cycle so that $CPI^{ideal} < 1$

Reducing instruction dependency stalls is achieved through advanced **compiler rescheduling** and hardware **dynamic rescheduling** (out-of-order execution).

Improvement of floating point performance is achieved by processing FP instructions in parallel in special **vector ALUs**.

As seen in chapter 6 (slide 62), **branch prediction** reduces branch stalls.

Cache latency can be reduced by **prefetching** cache blocks to the CPU based on address prediction.

With **thread level parallelism** on multicore CPUs, code can be divided into independently executing threads further reducing CPI^{ideal} and data dependency.

In order to make sense of these developments, we distinguish four types of parallelism:

Pipelining

Instruction I_{n+1} begins before I_n completes as in DLX so that 1 instruction completes every clock cycle τ. Therefore $R = 1 / \tau$ instructions complete every second.

Superscalar

This type of CPU contains $M > 1$ copies of the pipeline operating in parallel. M instructions start on same clock cycle and ideally M instructions complete on every clock cycle.

Superpipelining

This type of CPU divides the pipeline into smaller stages and each stage performs less work. Less work means a shorter clock cycle $\tau' < \tau$, and therefore a higher clock rate $R' > R$. Now $R' > R$ instructions complete every second.

Multiprocessor

In this type of CPU, usually a multicore chip, $N > 1$ program sections run on N processors in parallel. Therefore the overall program runs in less time.

The DLX is the model of a scalar RISC **pipelined** processor. As we have seen, 1 instruction completes on every clock cycle in the idea case and so $CPI^{ideal} = 1$.

Slides 6 – 9 Vector Processors

Vector processors are special ALUs that implement the SIMD execution model — a single instruction is performed in parallel on multiple data operands. SIMD is very useful for graphics and audio processing where typically one particular operation must be performed on a large array or data structure. Slide 7 shows the execution diagram for an SIMD ALU added to the DLX. A parallel load copies 4 memory words (16 bytes) to register P1 in one CC. Another parallel load brings 4 more words to P2 in 1 CC and a parallel add performs addition in 1 CC (following the load-to-alu stall). A loop over a large data array will run 4 times faster on this SIMD vector processor than a standard sequential loop in the ALU.

Examples of commercial vector processors are listed on slide 8. Slide 9 shows an SIMD example for an Intel x86 CPU. The prefix ASM means that the compiler passes the line of assembly code through as machine code.

Slides 10 – 11 Cache Prefetch

A limitation of the SIMD code example is that the sequential data loads will cause many compulsory cache misses. The short loop shown on slide 10 performs 1024 x 1024 SSE SIMD operations on 16-byte operands — a total of 16 MB of data loads. Since the L1 cache block in a Pentium 4 CPU is 64 bytes which is four 16-byte SSE operands, there will be an L1 miss every 4 data accesses.

Cache prefetch is a method for bringing data blocks to cache before a cache miss. The prefetch instruction initiates copying of 2 blocks to cache on execution while the SSE is executing instructions on the previous cache blocks. The data blocks will be in cache by the time the SSE requires access to them, hiding the latency associated with the cache miss (the cache update latency is not removed — it is "hidden" by being run in parallel to useful work).

Slides 12 – 18 P6 Superscalar with Dynamic Rescheduling

In a superscalar processor, elements of the pipeline are duplicated. The Intel Pentium processor (1993) as built around 2 complete copies of the 5-stage Intel 486 pipeline (1989), but this method had significant limitations. The P6 architecture model instead duplicates only the execution units (ALUs, Load, Store, FPU) and uses dynamic rescheduling to allocate instructions to the ALUs. All Intel x86 processors have used a version of this architecture since the Pentium II (1997). The machine language for x86 processors conforms to the Intel IA-32 ISA, which is a CISC type language. Internally, instructions are converted to a RISC type language for execution. P6 can be described as a RISC ISA with a silicon-based CISC adaptation layer for accepting IA-32.

The stages of execution in the P6 architecture are:

Fetch/Decode

The first stage fetches multiple Intel IA-32 instructions from memory per clock cycle and converts these CISC type instruction to several (1 to 6) RISC-type instructions called **micro-ops**.

Instruction pool

The micro-ops are copied to an instruction pool (a large instruction register) until they are ready for execution. The instruction pool is also called a Reorder Buffer (ROB).

Scheduler

A scheduler performs **out-of-order dynamic rescheduling**. It scans the instruction pool for instructions in that are **ready** — their source operands are already known. The scheduler issues these micro-ops to parallel execution in the various execution units: ALU, FPU, Load, Store. After execution, the finished micro-ops are returned to instruction pool together with their execution results.

Retirement

The finished micro-ops are committed to state (written to registers and memory locations) in their original order in the program (in-order write back). From the programmer's point of view, the program executes and updates architectural state in the original program order.

Dynamic Scheduling

The dynamic scheduling is determined by a process called **scoreboarding**. Each instruction is labeled by a status field:

NR	Not Ready	At least one source operand is not available
R	Ready	All source operands are available
X	Executed	Instruction executed but the destination operand is not available
F	Finished	Instruction was executed and all destination operand(s) are available

Only instructions marked **ready** can be executed, according to a scheduling policy that depends on the hardware organization.

The scoreboard is updated after each clock cycle. Completed instructions are marked **finished** and instructions are marked **ready** as source operands become available.

Slide 14 shows a scoreboard for the standard DLX. Only the EX stage is shown because the pipeline order is well known. In CC1 the first LW is executed and marked X (not F because the loaded operand is not available until the next clock cycle — this is the Load-to-ALU stall in the DLX). The other LW instructions are ready, but cannot be executed because the DLX uses program-order scheduling. On CC2 the first LW is marked Finished and so the ADD is marked Ready. The program continues in this way, executing as expected.

Slide 15 shows a scoreboard for a **DLX that permits dynamics execution**. Instructions are executed as they become ready — the three LW instructions are executed on CC1 to CC3. On CC2 the first ADD becomes Ready and so it is executed in CC4. Since there is no ALU-to-ALU stall the ADD is marked F in CC4. This makes the first SW ready. The SUB R3 instruction became ready next and is marked F in CC5. The program continues in this way — in the optimized scheduling that would have been generated by an optimizing compiler.

Slide 16 shows a scoreboard for a **P6** that permits dynamics execution. Instructions are executed in program order if they are ready. Only 1 load and 1 store is permitted per CC, along

with 2 ALU and 2 FPU instructions. The first LW executes in CC1, followed by another LW and ADD in CC2. The program finishes in 5 CC instead of 12 CC on the standard DLX.

Slide 17 summarizes the P6 execution. Three IA-32 CISC type assembly instructions are interpreted to 9 RISC type micro ops. These micro ops are executed in 5 CC as the scheduler identifies ready instructions and issues them to execution units.

Slide 18 shows the utilization of executions units (EU) in each clock cycle. We see that most EUs are idle in most CCs. The program executes in the minimum number of sequential cycles but with low hardware utilization. In order to obtain higher ILP it is necessary to achieve higher utilization of the execution units, and this requires a larger pool of independent instructions.

One method to find more instructions without data dependencies is through **speculative execution**, which uses branch prediction to find instructions that are likely to run and to execute them before the actual program flow is determined. This is a useful approach when the predictions are good — otherwise the CPU performs useless work that must be cancelled. Another way to increase the pool of independent instructions is through multithreading. Instructions from different threads are inherently independent and so running multiple thread on a single processor can produce many independent instructions.

Slides 19 – 23 Deep Superpipeline

In superpipelining, each pipeline stage is divided into 2 smaller stages. Each new stage does half the work in half the time and so the new stages finish their work in half the time. Thus, it is possible to divide the clock period τ in half which doubles the clock speed. As long as CPI^{ideal} does not change, this means that the run time should divide in half, producing a speedup of 2.

The problem with superpipelining is that some activities cannot be effectively split in half, for example a simple ALU operation. Also, some operations **do not scale in time**. External events take a fixed time so that running a faster CPU clock means that more stall cycles are wasted while waiting for cache updates, branch penalties, page faults, and so on.

As an **example**, the Intel Pentium III was designed with a 10 stage pipeline that ran at a clock speed up to about 1.5 GHz. The successor Pentium 4 as designed with a 20 stage pipeline at a clock speed up to about 4.0 GHz. Based on run time considerations we expect that a 1.5 GHz processor will be 50% faster than the same processor at 1.0 GHz. Since the instruction set is nearly the same for the two processors, it was surprising to find from measurements on SPEC CINT2000 that the 1.5 GHz Pentium-4 was only 20% faster than the 1.0 GHz Pentium III. Analyzing published statistics on the Pentium III one can estimate its total $CPI^{stall} = 0.2$. It is then possible to conclude that the Pentium 4 has $CPI^{stall} = 0.5$, larger than the older CPU. These extra stalls are probably a consequence of doubling the pipeline length.

In order to handle the increased CPI^{stall} Intel introduced **Hyper-Threading**. The hyper-threaded CPU contains two copies of architectural state (registers, stack pointers and program counter) but only one execution core (complete set of EUs). The OS sees two sets of registers (in particular two program counters requesting instructions) and so it looks like two CPUs. The OS assigns threads to both CPU 0 and CPU 1. Since both CPUs issue instructions to a shared execution core, they alternate operations. As long as there is no data stall in either thread, CPU 0 and CPU 1 request instructions on alternate clock cycles. If there is a stall in one thread, the other CPU continues to issue instructions on each clock cycle until the stall ends. Both CPUs keep working on most clock cycles. A stall in a hyper-threaded CPU requires a stall in both threads. Based on estimates from performance statistics, it is possible to estimate that hyper-threading provides a 20% speedup, which is precisely the measured result published by Intel.

Slide 24 Intel Nehalem Micro-Architecture

As an example of the P6 slide 24 shows the Nehalem microarchitecture (2008) developed for the i7 processor.

L1 instruction cache

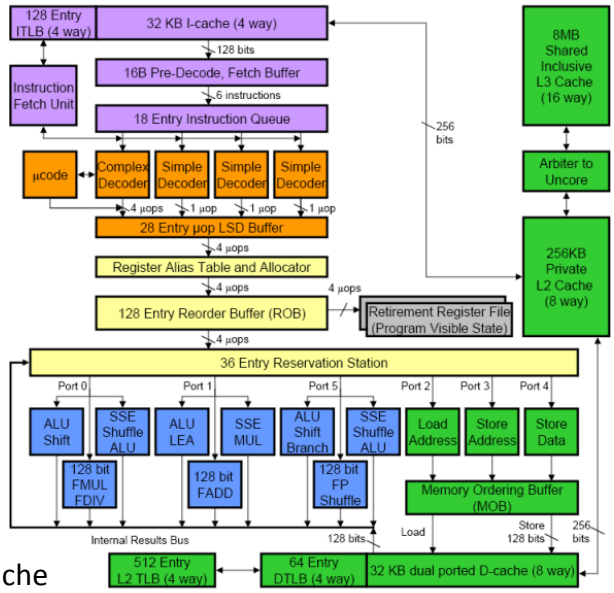
Fetch/decode Unit

LSD Buffer

Instruction Pool (ROB)

Execution Units

L1 data cache



L3 cache

L2 cache

The **LSD buffer** is a Loop Stream Detector. The IA-32 instructions within a loop are not re-fetched and re-decoded in each loop iteration. The decoded instructions within a loop are stored in the LSD in their order of dynamic execution, so that each loop simply recalls the stream of micro ops run in the previous iteration, in the same order.

Slides 24 – 27 Parallel Processing

In parallel processing, instructions are divided among processor cores and executed in parallel (at the same time). Running a program on N-processors divides the number of clock cycles by N for instructions that can be parallelized. **Amdahl's equation** in this case is derived on slide 24.

Slide 25 shows statistics on speedups for 2 and 4 cores. As stated in slide 22, hyper-threading for one CPU provides a speedup of S = 1.2, a 20% improvement. Without hyper-threading we see a speedup of 1.7 for 2 cores and 2.6 for 4 cores. Inverting Amdahl's equation for these values, we find that the proportion of the work that can be parallelized is F_P = 80%.

The difficulty in parallel processing led to a dramatic fall in research in the field after 1995. At that time it was generally believed that during the time required to write a parallel version of a large application with a speedup of 2, the clock speed of available CPUs would also rise by 2, making the investment in parallel processing not worthwhile. Slide 26 shows that the number of research papers in the field grew quickly from 1975 to 1995 and then fell off quickly after 1995. The return to parallel computing is now motivated by lack of other options. Chips with large numbers of cores are much easier to build than chips with faster clock speeds.

Parallel computers can be divided into two basic categories:

Shared memory

The coordination between threads (called **interprocess communication**) is performed through write/read operations to shared memory locations. The system has a single address space shared by multiple cores. Sequential memory coherence is enforced by cache snooping (the interprocessor bus imposes correct write/read order). This creates a cache coherency overhead in processing time.

Message passing system

Interprocess communication is performed by sending and receiving structured messages between cores. These messages send and request data or status information. Sequential coherence is enforced by the message content and message order synchronization. There is no snooping or snooping overhead but message management contributes overhead to processing time. Message passing is generally used for very large distributed systems using the MPI (message passing interface) API.

Slides 28 – 34 Shared Memory

Shared memory parallel computing does not scale well to large systems and is usually used for multicore systems. One or more physical microprocessors each contain private cache, architectural state (registers, including stack pointers and program counter) and execution core (integer ALUs, FPUs, vector processors, memory access). The OS assigns a thread to each processor that runs independently. On a long stall (page fault) a CPU can switch threads.

On multicore CPUs, there is usually a private L1 cache for each core and a shared L2 cache for all cores.

A convenient API for shared memory parallel programming is **Open MP** (supported by gcc 3 and newer versions). Open MP (OMP) supports shared memory applications in C/C++ and Fortran, providing directives for explicit thread-based parallelization. OMP uses the Fork-Join Model in which a master thread (consumer thread) initiates as a single thread and executes sequentially until a parallel construct is encountered. It then forks a team of parallel producer threads. When the parallel section completes, a join operation combines the results of the threads and the master thread continues. Parallel teams can be nested within parallel sections. Slide 31 shows a schematic parallel section within a C program and slide 32 shows a parallel "Hello, worlds" program in which each thread prints "Hello, world from thread number __". The master thread prints the total number of threads.

Slide 33 shows a **parallel for** loop in which 12 loop iterations are split among 3 threads that each perform 4 loop iterations in parallel. This form of parallelization is called **data decomposition**.

Slide 34 shows a **functional decomposition** using the **sections** structure that assigns different programming tasks to each thread.

Slides 35 – 38 Message Passing

Slide 35 shows an example of parallel programming on a message passing system. Each of 4 CPUs performs one part of a scalar product calculation in parallel. Then P0 sends a message to P1 with its result and P2 sends its result to P3. P1 adds the result from P0 to its own, and sends the sum to P3. P3 receives the result from P2 and adds it to its own result. Then P3 receives the sum from P1 and adds it to its sum, forming the final result. The coherence of results is maintained by information in the messages including the sender ID and a timestamp.

Message passing systems define **collective operations**: scatter, gather and reduce. Scatter distributes data among several CPUs and gather collects results from several CPUs. Reduce performs an operation on the results from several CPUs.

A standard API for message passing is the message passing interface (MPI). An example of an MPI Hello, World program is shown on slide 38.