Chapter 9

COMPUTER ARCHITECTURE

Joshua J. Yi[1] *and David J. Lilja*[2]
[1]Freesale Semiconductor Inc.
[2]University of Minnesota

1 INTRODUCTION

Originally proposed in 1945 by John von Neumann, the von Neumann architecture has become the foundation for virtually all commercial processors. von Neumann machines have three distinguishing characteristics: 1) the stored-program concept, 2) the partitioning of the processor into different functional components, 3) and the fetch-execute cycle.

The key idea behind the stored-program concept is that the series of instructions that form the program are stored in processor-accessible memory. By contrast, for processors that do not utilize the stored-program concept, the instructions of the program need to be fed into the processor as the program is running or the program needs to be hard coded into the processor. Storing the program in memory where the processor can easily access it is obviously more efficient than feeding in each instruction while the program is running. Also, reprogramming a stored-program concept processor is as simple as loading the next program into memory, which is more flexible than physically reprogramming the processor.

The second key characteristic of von Neumann architectures is that the processor is partitioned into components for input, output, computation, and control. Figure 9.1 shows how these components are connected together. The input and output components allow the processor to communicate to the user through other parts of the computer. For example, the processor receives information from the user through the keyboard and mouse while displaying information to the user through the monitor. The arithmetic-logic unit (ALU) is the component in the processor that actually does the computations. Computations can be divided into two categories: arithmetic and logical. Examples of the former include addition, subtraction, multiplication, division, etc. for integer and floating-point (real) numbers; examples of the latter include AND, OR, XOR, NOT, etc. In current-generation processors, the ALU is not a single monolithic

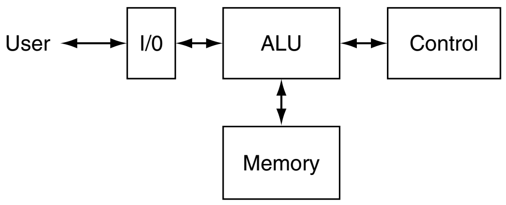

Figure 9.1. Basic components of a von Neumann architecture

component. Rather, multiple, distributed functional units perform its tasks; this partitioning decreases the overall execution time of each type of operation. Finally, the purpose of the control logic is to coordinate the flow of instructions and data between the different components of the processor by producing a sequence of signals that synchronizes the operation of each of the processor's components with respect to the other components. The control unit is necessary to ensure correct execution of the fetch-execute cycle, which is the third and final characteristic of a von Neumann architecture.

As its name implies, the fetch-execute cycle consists of two steps: instruction fetch and instruction execution. Since the program is stored in the computer's memory, to fetch an instruction, the processor must first retrieve each instruction from the computer's memory before the instruction can be executed. To retrieve the proper instruction, the processor sends the value of the program counter (PC), which holds the memory address of the next instruction, to memory, which returns the instruction stored at that memory location. After receiving that instruction, the processor calculates the address of the subsequent instruction and stores it into the PC. Usually, the address of the next instruction is simply the address of the instruction immediately following the current instruction. However, due to branch and jump instructions (which are the result of the function and subroutine calls, IF statements, etc.), the next instruction may not be the next sequential instruction, but will instead be located somewhere else in memory. Storing the address of the next instruction into the PC completes the instruction fetch part of the fetch-execute cycle.

The other half of the fetch-execute cycle, instruction execution, consists of several smaller substeps. The first substep is instruction decode, which occurs immediately after the instruction is fetched. In this substep, the decode logic analyzes the instruction to determine what kind of instruction it is (add, multiply, AND, branch, load, store, etc.), how many input operands there are, and where the input operands come from. After the processor decodes the instruction, it first gathers the values of the input operands, as specified by the instruction. For example, before the processor can compute the result of "1+2," it first needs to retrieve the values of the two input operands (1 and 2) from the specified registers or memory locations. In the next substep, the processor executes the instruction. In the case of arithmetic and logical instructions, the processor computes a new output value. In the case of load and store instructions, the processor accesses memory to either retrieve a value from memory (load) or write a value to memory (store). And in the case of branch instructions, the processor determines whether the branch condition is true or false and then calculates the memory address of the next instruction. Table 9.1 summarizes the action of these three types of instructions. Finally, in the last substep, the result of the instruction is stored into a register or a memory location so that is it available for the next instruction. After finishing execution, the processor fetches the next instruction and restarts the fetch-execute cycle all over again.

To summarize, instruction fetch retrieves the next instruction that the processor will execute, and in instruction execution, the processor performs the work that is specified by that instruction. By repeatedly fetching and executing instructions, the processor executes a program.

Although proposed over 50 years ago, the three fundamental characteristics of a von Neumann architecture, the stored-program concept, partitioned processor components, and the fetch-execute cycle, still remain the foundation of modern-day processors.

2 RISC VERSUS CISC

It is important to realize that the characteristics of a von Neumann architecture specify only how the processor is organized and how it operates from a functional point of view. As a result, two processors could have very different implementations, but both could still be von Neumann architectures. Given this freedom of implementation, computer architects have proposed two implementations that represent very different design philosophies. The first approach is known as the reduced instruction set computer, or more commonly by its acronym RISC. The second implementation is the complex instruction set computer, or CISC.

At heart of the difference between these two design philosophies is the processor's instruction set architecture (ISA). The instruction set is the set of assembly-level instructions that the processor is capable of executing and the set of registers that are visible (directly accessible) to an assembly-language programmer.

2.1 RISC: Reduced Instruction Set Computers

The basic design philosophy for RISC processors is to minimize the number and complexity of the instructions in the instruction set, in addition to defining uniform-length instructions. Adding more complex or nonuniform instructions into the processor's instruction set makes it more difficult for the processor to execute efficiently, since each of those instructions may have its own individual idiosyncrasies that may require specialized hardware in order to execute those cases. A processor can execute instructions much more efficiently when they are simple and uniform in length, since less complex "one-size-fits-all" hardware can be used for all instructions.

In addition to reducing the complexity of the hardware, minimizing the number of instructions in the processor's instruction set has the effect of reducing the

Table 9.1. Actions of the three main instruction types

Instruction Type	Action
Arithmetic and Logical	Computes new results
Load and Store	Reads from and writes to memory
Branch	Checks condition, determines next instruction

bus widths between internal processor components. Since each instruction has its own unique identifier, known as the *opcode*, adding additional instructions to the processor's instruction set may result in needing to add additional bits to the opcode. Adding more bits to the opcode may increase the number of bits needed for each instruction, which in turn increases the width of each internal bus.

Using simple and uniform-length instructions has two consequences. First, only load and store instructions are allowed to access memory; this simplifies the specification and execution latency of the instruction set's arithmetic and logical instructions. Instead of allowing an add instruction to directly add the values in two memory locations, two load instructions are used to first load those values into the processor's registers before the add instruction can execute. After computing the result, the add instruction has to use a store instruction to write its result to memory. This type of architecture is known as a "register-to-register" architecture, since all instructions with the exception of loads and stores can only read their input operands from and write their output values to the register file. Obviously, as compared with allowing each arithmetic or logical instruction to directly access memory, using load instructions to first load the values for input operands and then using a store instruction to store the output value requires three additional instructions (two loads and one store). While this increases the total instruction count when a program executes, it does not change the amount of "work" (steps necessary to execute the program) that the processor needs to do. Since this approach reduces the complexity of the hardware, this approach may still allow the processor to execute the program in less time than a CISC processor. In other words, it is easier to design hardware that executes a few types of instructions over and over again instead of designing hardware to execute many different types of instructions just a few times.

The other consequence of executing simple and uniform-length instructions is that there are fewer ways for load and store instructions to generate the address that is used to access memory. Since these ways of accessing memory are relatively simple, the compiler may need to insert additional instructions to help generate the correct address. For example, when traversing a linked-list in C, if more complex addressing modes were allowed, each load instruction could potentially retrieve the base address for the next link in the list. In a RISC processor, two loads are needed to retrieve the base address of the following link in the list.

In summary, since each RISC instruction is relatively simple and is uniform in length, programs compiled for a RISC processor contain additional instructions to move data between the processor and memory, to support more complex addressing modes, or to execute pieces of more complex tasks than a CISC processor. These additional instructions obviously increase the number of dynamic instructions that the processor executes, in addition to increasing the size of the compiled program. On the other hand, since all the instructions are simple and of uniform length, the hardware for the RISC processor is relatively simple and therefore more efficient, i.e., has a higher clock rate than the equivalent CISC processor. Simply stated, RISC processors trade off the execution of a larger number of instructions for a faster clock frequency.

2.2 CISC: Complex Instruction Set Computers

The basic design philosophy behind a CISC processor is nearly opposite to that of the RISC processor. First, instead of using several simple instructions to accomplish a single task, a CISC processor may use only one or two more complex instructions. Second, instead of having a set of relatively simple, uniform-length instructions, the instruction set for CISC processors consists of many complex instructions that are nonuniform in length and have multiple addressing modes. Third, instead of allowing only load and store instructions to access memory, in a CISC processor, arithmetic and logical instructions can access memory directly. As a result of these differences, a CISC processor typically executes fewer instructions to run the same program than the RISC processor does. Also, the size of the compiled program for the CISC processor, in terms of bytes, is also smaller than the size of the RISC program.

Obviously, these three differences have a very significant effect on the actual implementation of the hardware. Since each CISC instruction is much more complex than its RISC counterpart, the hardware needed to execute each CISC instruction is correspondingly more complex. In general, increasing the complexity of hardware decreases the speed at which the hardware executes instructions. As a result, the clock frequency of CISC processors is typically lower than the clock frequencies of RISC processors. Since each CISC instruction does more work than does a RISC instruction, each CISC instruction takes more time, as measured in clock cycles, to execute. Therefore, not only is the clock rate of CISC processors slower than that of RISC processors but also it typically takes more clock cycles to execute a CISC instruction than for a RISC instruction. However, the trade-off is that one CISC instruction does the same amount of work as several RISC instructions.

In summary, the design philosophy of CISC processors is to support very complex instructions that can be nonuniform in length. The upsides of this design philosophy are that each instruction does a significant amount of work and that the total size of the program is smaller. The downsides are that it takes more clock cycles to execute each instruction and that the hardware is very complex and consequently slower than the equivalent RISC processor.

2.3 Performance Analysis of RISC versus CISC

Although the previous two sub-sections compared RISC and CISC processors somewhat indirectly, this section uses the formula below to directly compare the performance of these two processors. The time required to execute a program is summarized below (see Eq. 1).

$$T_e = n * \text{CPI} * T_c \tag{1}$$

T_e is the total execution time of the program. n is the total number of dynamic (executed) instructions in the program, CPI is the average number of clock cycles needed to execute each instruction, and T_c is the time per clock cycle.

As this formula shows, the total execution time of the program depends on the number of instructions that the processor has to execute, the average number of clock cycles that each instruction takes, and the amount of time in a clock cycle (i.e., the reciprocal of the clock frequency). Therefore, to reduce a program's execution time, a computer architect can 1) reduce the number of instructions that the processor executes, 2) reduce the average number of clock cycles that it takes to execute each instruction, and/or 3) decrease the time per clock cycle (increase the clock frequency).

Since RISC processors execute more instructions than CISC processors do, the value of *n* is higher for RISC processors. However, the corresponding tradeoff is that each RISC instruction takes fewer clock cycles when executing the same program, which means that the CPI for RISC processors is lower. Finally, since the hardware for RISC processors is less complex, the clock period for RISC processors also is typically lower for a given technology.

Ultimately, the key question is, which design philosophy is the better approach? The answer is usually RISC, and the reason is called pipelining, which is explained in more depth in the following section. Due to its design philosophy of simple and uniform-length instructions, RISC processors benefit more from pipelining than the typical CISC processor does. Since pipelining is more difficult to implement on a CISC processor, and since it yields lower performance benefits for a CISC processor, RISC processors have evolved into the principal design philosophy used in the design of most current processors.

3 EXPLOITING PARALLELISM: PIPELINING AND MULTIPLE INSTRUCTION ISSUE AND EXECUTION

3.1 Pipelining

To reduce a program's execution time, computer architects need to either decrease the number of instructions that the processor executes, reduce the CPI of each instruction, or reduce the clock period. However, since the number of instructions in the program cannot be reduced at run-time by the hardware and since the clock period is limited by the minimum transistor width, the only viable option for computer architects to reduce the program's execution time is to reduce the CPI. Since it is very difficult to directly decrease the CPI of any individual instruction, the principal method to decrease the processor's CPI is to increase the number of instructions that are executing concurrently, i.e., executing instructions in parallel.

For example, assume that it takes a processor 5 clock cycles to fetch and execute an add instruction. This corresponds to a CPI of 5 cycles for that instruction. Then also assume that the multiply instruction that immediately follows the add instruction takes another 5 cycles. When these two instructions execute sequentially, i.e., one after another, the add instruction finishes after 5 cycles. In the next cycle, cycle 6, the multiply starts and then finishes 4 cycles later, in cycle 10. Therefore, in the case of sequential execution, the average CPI for these two instructions is 5 cycles.

On the other hand, assume that the multiply instruction starts executing one cycle after the add and that there are sufficient hardware resources to execute both instructions in parallel. The add instruction starts executing in cycle 1, while the multiply instruction starts executing one cycle later in cycle 2. In cycle 5, the add instruction finishes, while the multiply instruction does not finish until cycle 6. In this case, the average CPI for these two instructions is 3 cycles. In this example, executing two sequential instructions in parallel reduces the average CPI from 5 cycles to 3 cycles, or by 40%.

In the previous example, the execution of the add and multiply instructions was pipelined. The basic idea behind pipelining is that hardware resources should be as busy as possible. In a pipelined processor, the processor's hardware resources are organized into "stages." Each major task of instruction execution maps to one or more pipeline stages. Then, to execute an instruction, the instruction enters the pipeline and goes through each stage of the pipeline until its result is written to the register file or to memory and it exits the pipeline.

Within the fetch-execute cycle, the processor performs several tasks to execute an instruction. Generally, these are fetch, decode, issue and obtain input operand values, execute, and writeback (store the newly computed results back to the register file or to memory). Assuming that each of these tasks is organized into its own pipeline stage, and assuming that output buffers are placed after each pipeline stage to store the results of that pipeline stage, the result is a classical 5-stage pipeline. In this case, the first stage of the pipeline is the fetch stage, the second stage is the decode stage, and so on.

Each pipeline stage performs its task on only one instruction at a time, or in other words, there is only one instruction in each stage. Unless there are data and/or control dependences between instructions, each instruction spends only one cycle in each pipeline stage. (A more detailed explanation of data and control dependences is given in the following section, but for now, it is only necessary to understand that data and control dependences force delays between instructions, which increases the average CPI.) Since each instruction spends only a single cycle in each pipeline stage and since there is only one instruction in each pipeline stage, the number of cycles that it takes to execute a program with n instructions, without any data or control dependences, is (see Eq. 2):

$$\text{Total Cycles} = m + (n-1) \tag{2}$$

m is the number of pipeline stages. Assuming that the program is running on a processor with 5 pipeline stages, the first instruction in the program enters the pipeline at cycle 1 and then exits the pipeline after cycle 5. Therefore, the execution time of the first instruction is 5, or m, cycles. Then, since each following instruction starts one cycle after the instruction before it and finishes one cycle after it, once the first instruction finishes, since one instruction finishes executing every cycle, the remaining $n-1$ instructions require only an additional $n-1$ cycles. Consequently, an n-instruction program takes only $m + (n-1)$ cycles in order to execute the program completely.

By contrast, for an unpipelined processor, since each instruction takes m cycles to finish executing and since the following instruction cannot start executing until the previous one finishes, the total execution time is $n * m$ cycles. The speedup of a pipelined processor – as compared with an unpipelined one – for a very large program (n→ ∞), is (see Eq. 3):

$$\begin{aligned}\text{Speedup}_{n\to\infty} &= \frac{\text{Unpipelined}}{\text{Pipelined}} = \frac{n*m}{m+(n-1)} \\ &= \frac{1}{\frac{1}{n}+\frac{1}{m}-\frac{1}{n*m}} = \frac{1}{0+\frac{1}{5}-0} = \frac{1}{0.2} = 5\end{aligned} \qquad (3)$$

Therefore, for a 5-stage pipeline, when there are a very large number of instructions and when there are no data or control dependences, the execution time of a program that runs on a 5-stage pipelined processor is 5 times faster than the execution time of the same program on an unpipelined one.

In summary, the use of pipelining reduces the execution time of a program by overlapping the execution of different instructions. Pipelined processors exploit the parallelism inherent in programs to decrease the program's execution time. In the ideal case, when there are not any data or control dependences, a pipelined processor with *m* stages is *m* times faster than an unpipelined one. However, in typical programs, data and control dependences do exist and can severely degrade the processor's performance from its theoretical peak performance.

3.2 Data and Control Dependences

Data and control dependences are the by-products of relationships between instructions. There are three kinds of data dependences: output, anti, and flow. In a pipeline, these three dependences cause write-after-write, write-after-read, and read-after-write hazards, respectively, if the dependences occur between instructions that are in the pipeline simultaneously.

Output and antidependences are known as *name dependences*, since they are the result of two instructions sharing a register or memory location (name), but not with a producer and consumer relationship. In the case of an output dependence, both instructions write their output values to the same storage location, typically a register. This dependence is only a problem when both instructions are allowed to execute in parallel and where the second instruction may finish before the first. To ensure correct program execution, the first instruction needs to write its output value to the register before the second instruction writes its output value.

In an antidependence, the second instruction writes to the register that the first instruction needs to read from. To ensure correct program operation, the first instruction needs to read the value from the shared register before the second instruction overwrites the current value. Since output and antidependences are name dependences, assigning the second instruction to write to a different register will remove this dependence while maintaining correct program execution.

Flow dependences are the result of a producer and consumer relationship between two instructions. A flow dependence exists between the two only if the first instruction writes to a register from which second one reads. Therefore, to ensure that the second instruction executes correctly (computes its output value using the correct input values), the second instruction must delay its read of the shared register until after the first instruction writes to it. Since flow dependences have to be honored, they are known as "true dependences". Unfortunately, since the value of the producer flows directly to the consumer, the processor cannot execute both instructions in parallel. Instead, the second instruction has to wait for the first to produce its result. Proposing architectural techniques to mitigate the effect of these dependences and/or to completely break them are very common topics in computer architecture research and are discussed in Section 3.3 to Section 4.

The following segment of assembly code gives examples of output, anti, and flow dependences. In particular, an output dependence exists between instructions 1 and 3 (through register r1), an antidependence exists between instructions 1 and 2 (through register r2), and a flow dependence exists between instructions 2 and 3 (through register r2).

```
1. add r1, r2, r3       // r1 = r2 + r3
2. sub r2, r4, r5       // r2 = r4 – r5
3. mult r1, r2, r6      // r1 = r2 * r6
```

The problem with forcing two instructions to execute in a specific order is that it forces the first instruction to finish executing before the second can start or, at the very least, decreases the amount of overlap in the execution of the two instructions, either of which increases the CPI. From a pipeline point of view, dependences prevent two instructions from executing in adjacent stages. Instead, when a dependence exists between two instructions, pipeline "bubbles" (NOP or "no-operation") must be placed between the two instructions. The pipeline has "stalled" when it executes NOPs instead of instructions. Alternatively, instructions without any dependences can be placed between the two instructions.

Finally, it is important to state that data dependences also exist when two instructions are not back-to-back. That is, an output, anti, or flow dependence can exist between two instructions that are separated by several other instructions.

While data dependences are due to the fact that two instructions read from or write to the same register or memory location, control dependences stem from the fact that the target (i.e., the next instruction) of the control (i.e., branch) instruction is unknown until the branch instruction finishes executing. Consequently, the processor cannot fetch and start executing the next instruction in parallel with the branch until after it completes execution. This forces the processor to either fill the pipeline with other instructions or with NOPs.

When a pair of dependent instructions are in the pipeline together, the dependence between the two instructions can cause a hazard. In other words, the dependence between instructions evolves from being a potential problem to an actual one, i.e., incorrect program execution.

3.3. Multiple Instruction Issue and Execution: SuperScalar and VLIW Processors

Although pipelining can dramatically improve the processor's performance, a hazard between any pair of instructions can dramatically degrade a processor's performance, since the processor has to stall the pipeline until the first instruction either reads its input value from, or writes its output value to, the shared register or memory location. Although this ensures correct program execution, not only does this increase the average CPI, it also has the effect of preventing instructions that are not dependent on the first instruction from being fetched and/or executed. Therefore, to avoid this problem, higher performance processors have the capability of fetching and executing multiple instructions in the same cycle to avoid being stalled by a single data dependence. This allows the processor to extract parallelism in another "dimension" to further improve upon the base processor's performance. Since these types of processors can issue and subsequently execute multiple instructions in the same cycle, these processors are typically called *n*-way issue processors (e.g., 4-way issue, 8-way issue, etc.).

To clarify, pipelining reduces a program's execution time by allowing one instruction to start executing in every cycle. A multiple issue processor, on the other hand, further reduces the program's execution time by allowing multiple instructions to start executing in every cycle. In other words, a multiple issue processor duplicates the pipeline such that multiple pipelines operate in parallel.

To support the simultaneous issue and execution of multiple instructions, several changes and additions need to be made. First, hardware structures like the register file need to be multiported so that multiple instructions can read from and write to them. Second, the buses between hardware components need to be widened to accommodate the flow of additional instructions. Third, additional hardware needs to be added to ensure that the processor can operate at peak efficiency or will operate correctly. An example of the former is the register renaming hardware. Since output and antidependences can be removed by simply renaming the shared register, the processor temporarily retargets the second instruction to write to another temporary register. In this way, the first instruction is able to read from or write to the shared register without the possibility of a premature write from the second instruction.

The reorder buffer (ROB) is an example of a component that is added to the processor to ensure that the processor executes the program correctly. Since multiple instructions begin executing every cycle and since a multiple-issue processor is pipelined, in any given cycle, there are several instructions that are currently executing. Since some instructions may finish executing before a preceding instruction, the processor needs to ensure that those instructions do not write their values to the register file or to memory, since they could be overwritten by what should be a preceding instruction. To store output values that are not yet ready to be written to the register file or memory, the processor uses a ROB. This hardware structure holds the results of instructions until each instruction is ready to write its value to the register file or memory in the correct order.

Computer architects classify current-generation processors into one of two groups: superscalar processors and very-long instruction word (VLIW) processors. Both of these processors use pipelining and multiple instruction issue and execution to increase the amount of parallelism. The difference between the two is in how the instructions are scheduled, that is, the order in which the instructions are to be executed. In a VLIW processor, the compiler schedules the order in which instructions will execute based on several factors, including any data and control dependences between instructions, the number and type of available functional units, and the expected execution latency of each instruction. After determining a set of instructions that meets the compiler's scheduling criteria, the compiler groups these instructions together to form a superinstruction, the very-long instruction word. Since the compiler has already determined that each group of instructions is free of any dependences within the group, each bundle of instructions can be fetched, executed, and retired (finished) together.

Use of the compiler to perform the instruction scheduling reduces the complexity of the hardware, since there is no need for complex scheduling logic, which could decrease the hardware's speed. Furthermore, since the compiler determines the instruction execution schedule at compile time, the potential exists for the compiler to construct a better schedule than would be possible by using only hardware. The compiler has the advantage of having more time than the hardware does when trying to determine an optimal schedule, and the compiler can examine more instructions at a time. However, the big problem with static (i.e., compiler-determined) instruction scheduling is that run-time information, such as the program's inputs, is not available. Not having the actual inputs of the program available to the compiler can significantly limit the compiler's ability to statically schedule a program.

By contrast, a superscalar processor dynamically, i.e., at run-time, determines the order of execution for the program's instructions. More specifically, after the instructions are decoded, the processor examines the decoded instructions to determine which ones are ready for execution. (An instruction is ready to be executed after it has received its input operands. An instruction that is not ready for execution must wait for its producer instruction(s) to compute the corresponding input value(s).) Depending on the issue policy, the issue logic in the processor then selects a subset of the ready instructions and issues (sends) them to the functional units for execution. If the processor has an *in-order* issue policy, the processor issues only ready instructions from the oldest unissued instruction up to the first nonready instruction. If the oldest unissued instruction is not ready, then no instructions are issued in that cycle. By contrast, if the processor has an *out-of-order* issue policy, it issues as many ready instructions as possible, up to the issue width limit. Therefore, in the event that there are several ready instructions after the first nonready one, an out-of-order processor is able to issue those instructions out of program order, bypassing the nonready instruction to issue as many ready instructions as possible. Since ready instructions do not need to wait for older, unready instructions, out-of-order issue can yield significant performance improvements as compared with in-order issue.

On the other hand, the advantage that in-order processors have over out-of-order ones is simpler hardware design. Since the processor only checks the oldest few instructions, instead of all unissued instructions, less complex hardware is needed to issue the instructions. The trade-off is that ready instructions younger than the first nonready one cannot be issued that cycle. Consequently, a single nonready instruction blocks further instruction issue, which slows down the instruction execution rate. Although the out-of-order processor is able to issue any instructions that are ready, the issue logic hardware is much more complex, since the processor needs to examine all unissued instructions to find the maximum number of ready instructions that can be issued that cycle. This requirement obviously increases the complexity of the issue logic.

In summary, the fundamental difference between VLIW and superscalar processors is when the actual order in which the instructions are executed is determined – either statically at compile-time or dynamically at run-time.

3.4 The Memory Gap and MultiLevel Caches

One of biggest problems facing computer architects, now and in the future, is the "memory gap." The origin of this problem is that the speed of processors is increasing faster than the speed of memory is increasing. Therefore, as processor clock frequencies increase, the number of *cycles* required to access memory also increases. Since it may take a few hundred cycles to retrieve the data for a load instruction, the processor will eventually stop issuing any more instructions, since it cannot find more ready ones. Shortly after the processor stops issuing instructions, the processor finishes executing the last few issued instructions, and further instruction execution stops completely. Therefore, until memory returns the value of the load instruction, the processor is completely idle. For multiway issue processors, this phenomenon is especially problematic, since the processor cannot execute any more instructions for several hundred cycles while it is waiting for the results of a load instruction. Instead of executing *n* instructions per cycle, where *n* is the maximum issue width, for a few hundred cycles, the processor stalls, i.e., "instruction slots" are wasted. Obviously, if the processor has to stall frequently to wait for memory accesses, the execution time of the program will be much higher than if the processor did not have to wait for memory accesses. To further exacerbate this problem, in addition to the increasing memory gap, the issue width of processors is also increasing. This means that even more instruction slots will be wasted in the future as the gap between processor and memory speeds increases.

To combat this problem, computer architects add small, fast memory structures called *caches* between the processor and memory (RAM). Caches exploit *spatial and temporal locality* to improve the performance of the memory hierarchy. Due to spatial locality, the next memory reference is likely to access an address that is close, physically, to the last one. Due to temporal locality, the next memory reference is likely to access an address that was recently accessed. The memory references of typical applications exhibit both kinds of locality due to the use of loops and the linearly increasing value of the PC. Thus, to improve the performance of the memory hierarchy, caches store data around the most recently accessed addresses.

When the program begins execution, the cache is said to be "cold," or completely empty. As the processor requests data from different addresses, the cache stores the values at those addresses and nearby addresses. When the cache becomes full, selected entries in the cache are overwritten based on the organization of the cache and its replacement policy. When the processor requests the value for a memory address that is already in the cache, the cache can send the value to the processor, instead of forcing the processor to retrieve the value from main memory. This situation is referred to as a *cache hit*. The opposite situation is referred to as a *cache miss*. Since the latency of a cache hit is much lower than the latency for a memory access, the number of cycles needed to retrieve the value for that address will be much lower. Then, if a significant percentage of the memory accesses are cache hits, the average number of cycles needed for memory accesses – and, subsequently, the total program execution time – will be much lower. Generally, as the cache hit rate increases, the number of cycles required for a memory access decreases.

To balance the cost and performance benefits of cache memories, computer architects use multiple levels of cache. The level-1 (L1) cache, the level of cache closest to the processor, is the smallest but also the fastest. Since a cache exploits spatial and temporal locality, a small cache can still have a high cache hit rate but a low hit latency. A cache with a low hit latency but a high hit rate minimizes the memory access time. Due to the stored program concept, instructions are stored and fetched from memory. However, since the memory access patterns of instructions and data are very different, the L1 cache is usually split into two L1 caches, one for instructions and one for data, to further improve the cache hit rate. Each level of cache, L2, L3, etc., between the L1 cache and main memory is larger and can hold more data, but is slower. Each level of cache services the memory accesses that were missed in the caches between that cache and the processor, albeit with a higher access time.

To illustrate how multilevel caches can decrease the average latency of memory access, first assume that the memory hierarchy consists of an L1 data cache, a combined L2 cache, and main memory, which have hit latencies of 2, 10, and 150 cycles, respectively. Also assume that the hit rate for L1 is 80% and for L2 is 90%, while the hit rate for main memory is 100%. Then, for 1000 load instructions, 800 (1000 * 0.80) are L1 hits, 180 (200 * 0.90) are L2 hits, and the remaining 20 are memory hits. The average latency for these load instructions is (see Eq. 4):

$$\begin{aligned}\text{Average Latency} &= [(800 * 2) + (180 * 10) + (20 * 150)]/1000 \\ &= 6400 / 1000 = 6.4 \text{ cycles}\end{aligned} \tag{4}$$

By comparison, without the L1 and L2 caches, the average memory latency of these 1000 load instruction is 150 cycles (the access time of main memory), which will substantially increase the CPI. Therefore, as this example shows, by adding some small, fast caches to exploit spatial and temporal locality, computer architects are able to dramatically improve the performance of the memory subsystem.

3.5 Policies and Additions for High-Performance Memory

To further improve the performance of the memory hierarchy, computer architects have implemented two policies into the memory hierarchy and its interface with the processor core. The first policy, *load bypassing*, allows load instructions to bypass preceding store instructions in the order in which load and store instructions are issued to the memory hierarchy [11]. Since load instructions retrieve values from memory that are needed by the processor for further computations, decreasing the latency of load instructions has a larger effect on the program's execution time than does decreasing the latency of the store instructions. One way to decrease the effective latency of a load instruction is to issue it sooner than otherwise would normally occur. The one caveat to this policy is that the addresses for all store instructions preceding this load must be known, i.e., calculated, before the load is allowed to access the memory hierarchy. The reason for this is that a preceding store instruction may write to the same memory location as the load. If the load is allowed to skip ahead of a store that writes to the same memory location from which the load reads, then the load will retrieve the wrong value from memory since the store did not first write its value. If the address of the load differs from the address(es) of all of the preceding stores, then load is allowed to skip ahead of those stores.

A more aggressive version of this policy allows the load to access memory even when the addresses for all preceding store instructions are not known. This version defers the address check until after the load retrieves its value from memory. If none of the addresses of the preceding stores matches the address of the load, then the load forwards its value to the processor core. If the address of a preceding store matches the address of the load, the load discards its value. In the former case, the load instruction retrieves its value from memory a few cycles earlier than it could have if it waited for the address calculation of the preceding store instructions.

In the following example, instruction 3 can execute can before instructions 1 and 2, since the memory address of instruction 3 (A) differs from the addresses for instructions 1 and 2 (B and C). However, instruction 4 can bypass only instruction 2, since its address (B) matches the memory address of instruction 1.

```
1.  st B, r1        // B = R1
2.  st C, r2        // C = r2
3.  ld A, r3        // r3 = A
4.  ld B, r4        // r4 = B
```

When the address of the load matches the address of a preceding store – as is the case for instructions 1 and 4 – and if both addresses have been computed, then *load forwarding* can be used to improve the processor's performance [9]. With load forwarding, the value of the store is directly sent to the load. In the event that two preceding stores write the same address, the load instruction receives its value from the second store. Sending the results of the store to the load directly has three benefits. First, it allows to the load to execute before the store, even though the store precedes the load and accesses the same address. Second, since the load obtains its value directly from the store instruction, it does not have to wait until the store instruction has first written its value to memory before accessing the memory hierarchy to retrieve that value. Finally, since the load instruction does not need to access memory, the amount of traffic within the memory hierarchy is reduced.

In addition to the cache size, the other factor that affects the cache hit rate is its associativity. The associativity can be defined as the number of cache entries where a specific memory address can be stored. In a direct-mapped cache, each memory address can only be stored in one cache entry. On the other hand, in a fully associative cache, any memory address can go in any of the cache entries. Since many addresses map to the same cache entries, increasing the associativity increases the number of locations in which the data for a memory address can be stored, which decreases the likelihood that that memory address will be overwritten when the cache is full. Two issues limit the degree of associativity. First, increasing the cache's associativity requires additional hardware for comparators and multiplexors, although the capacity of the cache, as measured in bytes, does not increase. Second, due to this additional hardware, the access time of highly associative caches is higher than caches with the same capacity but a lower degree of associativity.

One very simple, yet highly effective, way of effectively increasing the cache's associativity is to use a *victim cache* [3]. A victim cache is a small, fully associative cache that stores cache blocks that are evicted from the L1 data cache. A cache block is a group of consecutive memory addresses that are moved in and out of the cache together. Cache blocks are evicted from the cache whenever empty entries in which an incoming block can be stored in cannot be found. Whenever a cache evicts a block, the next access to that block will require a higher access latency, since that cache block is present only in a level of cache that is further away from the processor. By contrast, when using a victim cache, evicted cache blocks remain in a level of cache closer to the processor. Although the victim cache is fully associative, its access time is similar to or lower than the access time of the L1 data cache, since it is so small. Use of a victim cache in parallel with the L1 data cache effectively increases the associativity of the L1 data cache since cache blocks can now be stored in the victim cache. Use of a victim cache in combination with the L1 data cache increases the hit rate of caches closest to the processor, which increases the processor's performance.

3.6 Branch Prediction: Speculative Bypass of Control Dependences

As described in Section 3.2, a control dependence stems from the fact that the instruction that should execute after a branch instruction is not known until after the branch has executed. However, waiting to the fetch the next instruction until after the branch has finished executing decreases the instruction throughput through the processor, which in turn increases the execution time of the program. Although the next instruction to follow the branch cannot be known with absolute certainty before the branch has started to execute, while the processor is waiting for the branch instruction to execute, the processor can predict the address of the branch target, i.e., the next instruction to execute, speculatively execute that instruction and the ones that follow it, and then verify whether the prediction was correct after branch finishes executing.

The processor component that makes predictions on the branch direction and target is the *branch predictor*. When the prediction is correct, the processor has successfully guessed which instructions will execute next and, consequently, the instructions that the processor had previously executed are correct. However, if the processor guesses wrong on which direction the branch will take next, then all the instructions that the processor speculatively executed are also wrong and need to be discarded. To return the processor to the correct state, the instructions that were speculatively executed need to be discarded and removed from the pipeline, and the processor needs to fetch and start executing instructions on the other path. The number of cycles that the processor needs to restore the processor state is known as the *branch misprediction penalty*. During this time, the processor is idle and not executing any instructions, which decreases the processor's performance. To maximize the performance of the processor, computer architects attempt to minimize the branch prediction penalty.

Two other key issues affect the processor's performance when using branch prediction. First, the number of stages in the pipeline affects how many cycles elapse before the branch predictor can verify the accuracy of the prediction. To verify the accuracy of the branch prediction, the branch predictor compares the result of the branch with the prediction. However, the longer the pipeline, the more cycles it takes for the processor to compute the result of the branch for verification—and the more cycles it takes for the processor to verify the prediction, the more cycles the processor spends executing instructions that will never be used. Since the number of stages in the pipeline directly affects the number of cycles that are needed to execute the branch, the number of stages consequently affects the processor's performance as it relates to branch prediction.

Second, the other issue is the branch prediction accuracy. The branch prediction accuracy is defined as the number of correct predictions divided by the total number of predictions. Since the length of the processor's pipeline and the branch misprediction penalty apply only when the branch predictor makes a misprediction, maximizing the number of correct predictions limits the performance degradation due to these two factors. Therefore, computer architects attempt to maximize the branch prediction accuracy.

Branch predictors consist of three main components: the branch history table (BHT), the branch target buffer (BTB), and some logic. The BHT is an on-chip table that stores the last *n*-directions for a few thousand branches. In the fetch stage, when a branch instruction is fetched from memory, the processor uses the branch's PC as an index into the BHT. The branch logic uses its algorithm and the branch's recent history to make a prediction as to whether the branch is taken or not. If the branch predictor predicts that the branch is not taken, then the next instruction that the processor will execute is the instruction that immediately follows the branch. If the branch predictor predicts that the branch is taken, then the branch logic uses the PC to access the BTB to quickly determine the address of the next instruction that is to be executed so that instruction can be fetched from memory. The BTB is a table that stores the addresses of recently-taken branch targets. Use of a BTB allows the processor to immediately start fetching the instruction at the predicted branch target instead of waiting for that address to be computed. After the branch executes, the processor updates the BHT with the direction of the branch and the BTB, if the branch is taken.

It is important to note that since the BHT is much smaller than the maximum number of entries that a PC could index, only a few bits from the least significant end of the PC are used to index the BHT; the remaining more significant bits are ignored. Consequently, since the entire PC is not needed to access the BHT, multiple branch instructions that have the same bit pattern for the BHT index will map to the same BHT entry. This situation is known as *aliasing*, and it can affect the branch predictor's accuracy since the branch history for another branch could be used to make predictions for the current branch instruction instead of its own history.

One simple branch predictor makes its predictions based on the last direction that is stored in the BHT for that branch, or another branch in the event of aliasing. If the last direction that the branch took was taken, then the branch predictor predicts that the branch will be taken again. The opposite prediction occurs when the branch was most recently not taken. After the branch executes, the BHT stores the direction that the branch actually took. Since only one bit is needed to store whether this branch was taken or not, this predictor is known as a one-bit predictor. While this branch predictor has the advantage of minimal BHT size and fair branch prediction accuracy, the major problem with it is that it tends to make mispredictions when entering and when leaving a loop.

For example, assume that the processor executes a loop with five iterations. For the first four iterations, the branch is taken; only the last iteration is not taken. Since the one-bit predictor immediately writes the most recent direction into the BHT, when entering the loop, the last direction that is stored in the BHT is not taken. Therefore, the branch predictor will predict "not-taken" for the first iteration when the direction is actually taken. After the first iteration, the BHT stores "taken" as the last direction for that branch and is subsequently able to make three correct predictions in a row for the next three iterations. However, for the fifth iteration, the branch predictor predicts "taken" when the branch is actually not taken. This results in another misprediction, and the branch predictor stores "not-taken" into the BHT, which will cause yet another misprediction when the branch executes the next time. This results in a 60% prediction accuracy due to mispredictions for the first and fifth iterations.

To solve this problem, a two-bit branch predictor can be used. The difference between the one-bit and the two-bit branch predictors is that the one-bit predictor changes its prediction in response to a single misprediction while the two-bit predictor requires two mispredictions to change its prediction. In the above example, the two-bit predictor would accurately predict the branch's direction for the first four iterations, making a misprediction only for the last iteration. Therefore, in this example, although the two-bit predictor requires twice the number of history bits in the BHT, it results in an 80% prediction accuracy, which is a very significant difference.

Other than one- and two-bit predictors, computer architects have proposed several other branch predictors to achieve higher branch prediction accuracies. One- and two-bit branch predictors are fairly accurate for floating-point programs where the branch behavior is relatively well behaved. But for integer programs, where the branch behavior is less well behaved, one- and two-bit branch predictors do not account for the effect that other branch instructions may have on the direction that the current branch will take and consequently have poor branch prediction accuracy.

In contrast, correlating and two-level predictors use the history of the most recently executed branch instructions to make a prediction. While there are several flavors and varieties of each, the basic operation for these two predictors is relatively similar. These branch predictors use a bit pattern that represents the taken/not-taken behavior of several recent branches as an index into a table of one- or two-bit prediction counters [10]. To store the direction of each of the m most recently executed branch instructions, these branch predictors use an m-bit shift register known as the *branch history register* (BHR). After a branch instruction finishes executing, BHR shifts the bits such that the oldest branch is overwritten and the youngest is stored on the other end of the shift register. The m-bits of the BHR are then used to index the *pattern history table* (PHT) that has 2^m entries. Each entry of the PHT is a one- or two-bit predictor that ultimately makes the branch prediction. It is important to note that basic versions of these predictors do not use the PC of the branch instruction, which may lead to aliasing in the PHT. To reduce the chance of deconstructive aliasing, variants of these predictors use at least part of the PC to index the PHT.

In summary, the basic assumption for these predictors is that whenever a series of branch instructions has the same history as another series, then the direction of the current branch can be predicted based on past behavior of the branch that followed each series of branch instructions.

After a processor jumps to and finishes executing a subroutine, it needs to return to the point in the program that called the subroutine. To accomplish this, the processor could use the PC of the branch instruction, which corresponds to the subroutine return, to access the BTB to determine what the next instruction is. The problem with this solution is that the subroutine could be called from several places in the program and that the calls may be interleaved. Therefore, the target (return) address could constantly change, depending on which place in the program called the subroutine. To avoid interrupting the instruction fetch process, computer architects have designed the *return address stack* (RAS) to store the address of the target instruction [4]. When a subroutine is called, the processor pushes the return address onto the RAS. If that subroutine calls another subroutine, or itself, another return address is pushed onto the stack. Then, when each subroutine has finished executing, the processor simply pops each return address off the RAS and resumes fetching instructions starting at the return address.

3.7 Branch Predication: Non-Speculative Bypass of Control Dependences

Although recently proposed and implemented branch predictors have become very complex – and accordingly require a large amount of chip area and dissipate a large amount of power – the control-flow of some branch instructions is so complex that they are hard to predict very accurately. To achieve higher branch prediction accuracy, which subsequently results in significantly higher processor performance, it is very important to accurately predict the direction of these difficult-to-predict branches. For reasons described above, difficult-to-predict branches severely degrade performance, since they interrupt the instruction fetch stream and since misprediction recovery requires several clock cycles.

One solution to this problem, called *branch predication*, is to simply fetch and execute instructions down both directions of the branch [7]. After the branch executes, the correct direction is known and instructions down the correct path are saved while instructions down the wrong path are ignored and discarded. To accomplish this, the branch instruction is converted to a compare instruction where the result of the compare is written to a *predicate register*. A predicate register is added as an input operand to each instruction down one path; instructions down the other path are assigned another predicate register. The value of the predicate register indicates whether the branch instruction was taken or not and subsequently whether the output values of that instruction should be saved or not. When the predicate register is set to zero, all instructions that have that predicate register as an input operand are discarded; meanwhile, the value of the predicate register for the instructions on the other path is 1. When the predicate register is set to 1, the instructions that use that predicate register are saved and eventually write their output values to the register file.

Since the processor executes instructions on both branch paths, the processor effectively predicts the direction of the branch with 100% accuracy. Therefore, why should branch predication not be applied to all branch instructions to achieve a 100% prediction accuracy? First, the cost of branch predication is that the processor must devote resources to executing some instructions that will be discarded. Therefore, the processor's execution rate is lower when the branch direction is unknown than after the branch direction has been resolved. Second, applying branch predication to all branch instructions means that even highly predictable branch instructions will be converted. This means that instead of making a high-accuracy prediction and then maximizing the rate of execution along that path, the processor sacrifices that high rate of execution for a much lower one to achieve a slight improvement in the branch prediction accuracy. Therefore, to maximize performance, branch predication should be applied only to difficult-to-predict branches.

In 2000, Intel began to ship the production version of the Itanium processor. One of the most notable features of this processor was its implementation of branch predication. Although initial academic studies suggested performance improvements of 30% or more, the performance improvement due to branch predication was a modest 2% [2]. Two key reasons were given to account for this discrepancy. First, there were several differences in the production and academic versions of the compiler and the hardware. One key difference was in the level of detail between the academic and production versions. For instance, the academic studies did not account for the effect of the operating system and factors such as the effects of cache contention and pipeline flushes. These relatively small differences tend to reduce the performance of the real machine. Second, the benchmarks that were used to generate each set of performance results differed. In the benchmark suite that was used on the production hardware, branch execution latency and the misprediction penalty accounted for a smaller percentage of the program's execution time than in the benchmark suite for the academic studies. Despite these differences and the difference in the performance results, the authors of [2] state that as the Itanium processor and its compiler mature, the performance impact of branch predication will increase.

3.8 High Performance Instruction Fetch: The Trace Cache

From a conceptual point of view, the instruction fetch and execute components of the processor exist in a producer-and-consumer relationship. The instruction fetch components, which includes the branch predictor and instruction cache, "produce" instructions by retrieving them from memory and placing them into a buffer known as the instruction fetch queue. The instruction execute components, which include the issue logic and the processor's functional units, "consume" the instructions by executing them and writing their results to the register file and memory. As the issue width increases, the rate at which the processor consumes instructions increases, which increases the processor's performance. However, to maintain the processor's performance as the issue width increases, the instruction fetch components need to produce the instructions at a similarly high rate or the processor's performance will suffer.

The problem with conventional instruction fetch mechanisms is that they can only fetch a single cache block from memory per cycle if the cache block contains a branch instruction that is predicted to be taken. When a cache block does not contain any branch instructions or when it contains a branch instruction that is predicted to be not taken, the next cache block is fetched next from memory. However, if the cache block has a branch instruction that is predicted to be taken, then the processor cannot fetch any more cache blocks until after the next block is brought into the processor. This severely limits the rate at which instructions can be fetched.

One solution to this problem is the *trace cache* [8]. The trace cache stores a trace of instructions that were previously executed together consecutively. Accordingly, the trace cache implicitly contains the record of which direction each branch instruction in the trace took. The trace cache is accessed in parallel with the L1 instruction cache, using the PC for the next instruction. When the processor finds a matching trace – one that has a matching set of predicted branch directions – in the trace cache, instructions are retrieved from the trace cache instead of from the L1 instruction cache. Otherwise, the processor fetches instructions from the instruction cache.

The advantage of using a trace cache is that by organizing the instructions into a trace, the instructions from multiple taken branches can be fetched from memory together in a single cycle. This gives the instruction fetch components the potential to meet the execution core's consumption rate. The disadvantage is that the processor designers must devote a substantial amount of chip area to the trace cache.

3.9 Value Prediction: Speculative Bypass of Data Dependences

As described in Section 3.2, in addition to control dependences, data dependences – register or memory dependences between instructions – also can severely degrade the processor's performance. The counterpart to branch prediction (speculative bypass of control dependences) is *value prediction*, which exploits *value locality* to improve the processor's performance.

Value locality is the "likelihood of the recurrence of a previously seen value within a storage location" in a processor [6]. In other words, value locality is the probability that an instruction produces the same output value.

Value prediction is a microarchitectural technique that exploits value locality. Based on the past values for an instruction, the value prediction hardware predicts what the output value could be. After predicting the output value, the processor forwards that predicted value to any dependent instructions – instructions that need that value as an input operand – and then speculatively executes those dependent instructions based on the predicted value. To verify the prediction, the processor executes the predicted instruction normally. If the prediction is correct, the processor resumes normal execution and can write the values of the speculatively executed instructions to the register file and memory. If the prediction is incorrect, then all the dependent instructions need to be reexecuted with the correct value.

It is important to realize that without value locality, value prediction would not be able to improve the processor's performance, since it would be virtually impossible to accurately choose the correct value for an instruction from 2^m different values, where m is the number of bits in each number (typically 32 or 64).

Last-value prediction is the simplest version of value prediction. Last-value prediction stores the last output value of each instruction into the value prediction table. Upon encountering the next instance of that instruction, the processor uses the last output value as the predicted value. For example, if a particular add instruction computed the output value of 2 last time, then when that add instruction next executes, the last value predictor predicts that the add will again produce an output value of 2.

While last-value prediction can yield reasonably high prediction accuracies for some instructions, its accuracy is very poor when it tries to predict the values of computations such as incrementing the loop index variable. Therefore, to improve the prediction accuracy of last-value prediction for these and similar computations, computer architects have proposed the stride-value value predictor. For this predictor, the predicted value is simply the sum of the last output value for that instruction and the stride, which is the difference of the last two output values. For instance, when the output value history for an instruction is 1, 2, 3, 4, 5, etc., the stride value predictor will predict that the next output values will be 6, 7, 8, etc. Note that when the stride value equals zero, the stride value predictor functions as a last-value predictor.

Although stride-value prediction has a higher prediction accuracy than last-value prediction, the two predictors are fundamentally the same. Consequently, for more complex output value patterns such as 1, 4, 7, 9, 1, 4, 7, 9, … 1, 4, 7, 9, etc., both value predictors have very poor performance. One value predictor that can accurately predict this irregular pattern is the finite-context method predictor. This predictor stores the last *n* output values for an instruction and then uses some additional logic to determine which of those *n* values should be used as the predicted value.

In summary, value prediction improves the processor's performance by allowing it to execute instructions earlier than would otherwise be possible, if the prediction is correct. This potential performance gain comes at the cost of prediction verification and a potentially very large value prediction table.

3.10 Value Reuse: Nonspeculative Bypass of Data Dependences

During the course of a program's execution, a processor executes many redundant computations. A redundant computation is one that the processor had performed earlier in the program. Any and all computations can be redundant. It is important to note that an optimizing compiler may not be able to remove these redundant computations during the compilation process, since the actual input operand values may be unknown at compile time – possibly because they depend on the inputs to the program.

Redundant computations affect the program's execution time in two ways. First of all, executing the instructions for redundant computations increases the program's dynamic instruction count. Secondly, these redundant computations affect the average CPI, since they produce the values for other instructions in the program (a flow dependence exists between these instructions and others). Unfortunately, while redundant, these computations need to be executed to ensure correct program operation. Consequently, the hardware cannot simply disregard these computations.

Value reuse is a microarchitectural technique that improves the processor's performance by dynamically removing redundant computations from the processor's pipeline [12]. During the program's execution, the value reuse hardware compares the opcode and input operand values of the current instruction against the opcodes and input operand values of all recently executed instructions, which are stored in the value reuse table (VRT). If there is a match between the opcodes and input operand values, then the current instruction is a redundant computation and, instead of continuing its execution, the current instruction gets its output value from the result stored in the VRT. On the other hand, if the current instruction's opcode and input operand values do not match those found in the VRT, then the instruction is not a recent redundant computation and it executes as it normally would. After finishing the execution for each instruction, the value reuse hardware stores the opcode, input operand values, and output value for that instruction into the VRT. Value reuse can be applied at the level of individual instructions or to larger units, such as basic blocks [5].

The key difference between value prediction and value reuse is that value prediction is speculative whereas value reuse is nonspeculative. Consequently, the predictions of the value predictor must be verified with the actual result of the predicted instruction, and recovery must be initiated if the prediction is wrong. By contrast, since the computation and inputs are known, the results for value reuse are nonspeculative and do not need to be verified, since they cannot be wrong.

While value reuse is able, through table lookups, to generate the output value of an instruction sooner than would otherwise be possible, two key problems limit its performance. First, to ensure that multiple instructions can access and retrieve their output values from the VRT within one or two cycles, the number of entries in the VRT has to be relatively low. Therefore, the VRT can only hold a small number of redundant computations. The second problem is that since VRT is finite in size and since it constantly stores the inputs and outputs of the most recently executed instructions, the VRT may eventually become filled with computations that are not very redundant. Therefore, instead of storing the redundant computations that are very frequently executed, which account for a large fraction of the program's execution time, the VRT may store redundant computations that are relatively infrequently executed and that have very little impact on the program's execution time.

3.11 Prefetching

As described in Section 3.4, the performance of the memory hierarchy is the result of two factors: the hit latency of the caches (or memory) and the hit rate of the caches. Since the hit latency is determined by how the cache is implemented, its size, associativity, and location (on-chip or off-chip), computer architects can only improve the hit rate to decrease the memory access time of load instructions. One such approach is a mechanism called *prefetching* [15].

What prefetching attempts to do is to retrieve a cache block of instructions or data from memory and put that block into the cache before the processor requests those instructions or data from memory, i.e., needs to use them. For prefetching to significantly improve the performance of the memory hierarchy, a prefetching algorithm needs to do two things. First, it needs to predict those address(es) for which the processor will access memory. Due to very complex memory access patterns that are prevalent in nonscientific applications, accurate prediction of which address(es) will be needed in the near future is very difficult. Second, for prefetching to be most effective, the prefetching algorithm needs to place the block of memory into the cache before the processor requests those instructions or data. However, due to wide-issue processors and very long memory latencies that are only getting longer, the prefetch algorithm must determine which block of memory to retrieve several hundred cycles or more before the processor actually makes that request. On the other hand, bringing the desired memory block into the cache far before it is needed may result in that memory block being replaced by another, higher-priority memory block. Therefore, the timeliness aspect of prefetching really means that the prefetched block needs to be brought into the cache as close as possible to when the processor will consume those values. Bringing that block into the cache too early or too late may not significantly improve the processor's performance.

Prefetch algorithms can be initiated either solely by hardware or with some assistance from the compiler. In the former case, the prefetching algorithm is completely implemented in the hardware. As a result, the hardware determines which addresses to prefetch and at what time. In the latter case, the compiler inserts prefetch instructions into the assembly code. Those instructions tell the hardware prefetch mechanism when to prefetch and for what to address to prefetch. For software prefetching, the compiler analyzes the assembly code to determine which load instructions will seriously degrade the processor's performance. For those instructions, the compiler then inserts the necessary prefetch instructions into the code at a point that it determines is sufficiently far away from the point in time when the processor will actually use that value.

One very well-known prefetching technique is *next-line prefetching*. When using next-line prefetching, after a cache miss, in addition to fetching the cache block that contains the address that caused the cache miss, the processor also fetches the next sequential cache block and places that block in a prefetch buffer, which is a small, fully associative cache. By fetching the next cache block, this prefetching algorithm is counting on the program to exhibit spatial locality and on addresses in the next cache block to be requested soon. Storing the prefetched cache block in a prefetch buffer reduces the amount of cache pollution, which is caused by bringing in blocks that will not be used before they are evicted or evicting blocks that will be used in the near future.

Finally, due to the increasing memory gap, designing and implementing more effective prefetching algorithms remains a very active area of research in computer architecture.

4 MULTITHREADED ARCHITECTURES: NEXT-GENERATION MULTIPROCESSOR MACHINES

4.1 Speculative Multithreaded Processors

Scientists and engineers, in an effort to decrease the execution time of their programs, commonly run their programs on multiprocessor systems. Ideally, after parallelizing the code, each processor can execute its portion of the program without having to wait for other processors to catch up or to produce values for it. A deeply nested loop, where each loop iteration does not depend on the value of a previous loop iteration, is ideal for parallelization, since it does not contain any cross-iteration dependences. While this situation is common for scientific floating-point applications, the loops in integer (nonscientific) programs typically have cross-iteration data dependences that make them very difficult to run on a multiprocessor system. These data dependences force the other processors in the system to stall until the processor running that previous loop iteration generates the needed value.

To address this issue, computer architects have proposed *speculative multithreaded processors* as a potential solution to allow integer programs to efficiently run on multiprocessor systems. A representative example of a speculative multithreaded processor is the Superthreaded Architecture (STA) [13]. In the STA, the compiler analyzes the program to determine which loops can be efficiently parallelized to decrease the overall program execution time. Since, at compile time, the compiler may not be able to determine whether a potential cross-iteration dependence will actually be one at run-time, the compiler flags that address. To ensure that those addresses that the compiler has flagged are handled properly at run-time, the STA uses an on-chip buffer called the *memory buffer*.

Other than the memory buffer, the only other additions to the base processor, which can either be an "off-the-shelf" superscalar or VLIW processor, are a little additional logic for interprocessor communication and for processor execution synchronization. Each processor is connected to two other processors via a unidirectional ring. As with typical multiprocessor systems, each processor has its own private L1 data cache but shares the L2 cache with the other processors. The memory buffer is a private cache.

When a program begins executing on the STA, only one processor is active; the remaining processors are idle, waiting for the program to reach a loop (parallel region). The start of the parallel region is denoted by a special instruction. After the active processor executes that special instruction, it forks off the next processor in the unidirectional ring and begins execution of its iteration in the parallel region. Meanwhile, the next processor copies the set of values that are needed for parallel execution and then forks off its own processor. This process repeats itself until all processors are executing an iteration of the loop.

When each processor in the system begins parallel execution, the processor allocates space in the memory buffer for each potential cross-iteration dependence that the compiler flagged. When a load instruction accesses the memory hierarchy, the memory buffer and L1 data cache are accessed in parallel, although the data can be present only in one structure. When the address for the load instruction is found in the L1 data cache, the processor continues execution as normal. However, when address is found in the memory buffer, the processor needs to wait until either another processor generates that value or another processor updates it, if the value is not already there. On the other hand, when a processor generates a value for an address that is found in the memory buffer, the processor forwards that value across the unidirectional ring to the other processors. Therefore, by using the compiler to flag potential cross-iteration dependences, the memory buffer to track and update the status of those dependences, and the unidirectional ring to pass values from processor to processor, the

STA architecture is able to parallelize and efficiently run programs that have cross-iterations dependences.

When a processor finishes its iteration, it checks to see if all processors that are executing a previous iteration are finished. If not, the processor stalls until they are finished. If so, then the processor writes its values back to memory. After a processor writes its values to memory, the state of memory is the same as if this iteration just finished executing on a uni-processor.

Finally, since there are an indeterminate number of iterations for some loops, it is not known until run-time which processor will be the one to execute the last iteration. To maintain high performance given this uncertainty, each processor keeps forking off another processor as if no uncertainty exists. When the processor that executes the last iteration detects that it is the last iteration, it kills all successor iterations running on the "downstream" processors. That processor then starts executing another sequential region of code while all other processors are idle. This cycle of sequential and parallel execution continues until the program is finished.

In conclusion, speculative multithreaded processors, such as the Superthreaded Architecture, allow multiprocessor systems to efficiently execute programs with many cross-iteration data dependences. To accomplish this, hardware like the memory buffer is added to the base processor to ensure that potential cross-iteration dependences are handled correctly.

4.2 Simultaneous Multithreading

When multiple programs are running on the same uniprocessor system, the operating system allows each program to execute for a certain amount of time before swapping that program out for another one. Before the next program can start running, the processor state of the current program must be saved to memory. Then after the processor state of the next program is loaded into the processor, the next program can begin executing. Obviously, repeated storage and loading of the processor state of each program adds extra overhead to the time it takes to execute both programs. On the other hand, running more than one program at a time may allow the processor the hide the latency of cache misses by running another program while the memory hierarchy services the cache miss.

Some operating systems and processors switch programs only when there are caches misses. This allows one program to efficiently execute the low latency parts of the code and then allows another program to run while high latency parts of the code are being serviced. Although this setup allows the execution of two programs to overlap, it still incurs the cost of saving and loading the processor state.

A hardware improvement on this approach is *simultaneous multithreading* [14]. A simultaneous multithreading (SMT) processor allows two or more programs, or threads, to simultaneously execute on the same processor. Therefore, instead of fetching only the instructions for a single program, an SMT processor fetches the instructions for multiple programs at the same time. Instead of decoding and issuing instructions from a single program, an SMT processor decodes, issues, and executes the instructions for multiple programs. Finally, instead of writing the results for a single program to the register file, an SMT processor writes the result of each program to its own register file to maintain proper program execution semantics. An SMT processor also replicates other base processor resources to support multiple hardware-based program threads.

Suppose that eight programs are running on a SMT processor. Of those eight programs, only two are active at any given time. To execute both programs, the SMT processor first needs to fetch instructions for both programs at the same time. After fetching instructions from both programs, the instructions execute on the processor as would the instructions for a single program, with the obvious exception that instructions for one program do not use the results from the other. Whenever a load instruction experiences a cache miss, the SMT processor stops fetching instructions for that program and starts fetching and executing the instructions for another program. Use of a round-robin fetch policy ensures that progress is made in executing each program.

It is important to note that the start-to-finish time of any one program running on an SMT processor will be longer than the start-to-finish time of the same program running on a conventional uniprocessor of similar resources, since the other programs compete for the processor's resources. However, the SMT processor is able to decrease the overall execution time of all *n* programs by executing them in parallel at a fine-grain level as opposed to executing them serially on the conventional uniprocessor machine.

SMT processors are able to decrease the execution time of multiple programs for three reasons. First, by allowing instructions from multiple instructions to execute simultaneously, the SMT processor can find more instructions that are ready to issue, since dependences, both control and data, do not exist between the instructions of two different programs. As a result, the SMT processor can reduce the average CPI across all programs. Second, by supporting multiple program execution at a very fine-grain level, the SMT processor is able to avoid the cost of storing and loading the processor state. Third, by swapping out each program after it incurs a cache miss, the SMT processor is able to hide the memory latency of a load instruction in one program by executing instructions in another program.

Finally, although their names are similar, simultaneous multithreading and speculative multithreading have several major differences. First, an SMT processor is a single, very wide-issue processor, while a speculative multithreaded processor is a multiprocessor system. Second, SMT processors decrease the overall execution time of multiple programs, while speculative multithreaded processors decrease the execution time of a single program. Given these differences, these two approaches could be combined together to form a multiprocessor system that can quickly execute a single program or multiple programs.

5 CONCLUSION: FUTURE TRENDS AND ISSUES

Over the past few decades, computer architects have improved the performance of processors in one of three ways: increasing the processor's clock frequency, executing multiple instructions simultaneously via pipelining, and executing multiple instructions in parallel via wide-issue processors. Although these approaches have significantly improved the processor's performance, at least two factors limit further performance gains purely by using these techniques.

First, as described in Section 3.4, the disparity in the rates of increase in the processor speed and the memory speed has led to a memory gap that will only widen in the future. To ensure that the processor is able to fetch an adequate number of instructions to feed the execution core and to ensure that data dependences between load instructions and other types of instructions do not become the bottleneck, computer architects need to find additional methods of decreasing the average memory latency. Compounding this problem is that memory bandwidth will increasingly become a limiting factor on any solutions [1]. As a result, instead of trading off memory bandwidth for memory latency, computer architects will need to find other solutions that decrease the average memory latency without dramatically increasing the memory bandwidth requirements. Solutions to this problem may include novel methods of prefetching data and instructions, different cache and memory hierarchy designs, and new technologies that reduce the memory latency.

Another problem that has already become a major one is the power dissipation of modern processors. Microarchitectural techniques such as branch prediction, prefetching, and value prediction are all speculative techniques that rely on predicting in what direction the branch will go, what instructions or data are needed next, and what values a particular instruction will produce, respectively, to improve the processor's performance. Although these techniques are very effective in improving the processor's performance, they also consume a lot of additional energy to do so. The extra energy that these and other speculative techniques consume increases the power consumption, which lowers the battery life of laptop computers or raises the temperature of the processor to dangerous levels. Consequently, for any performance enhancements, computer architects must balance increased performance with increased power consumption.

In conclusion, to maintain the phenomenal rate of improvement in microprocessor performance, computer architects need to implement the techniques that have been discussed in this chapter. Also, architects need to develop other techniques of improving the performance without exceeding power consumption goals.

REFERENCES

[1] D. Burger, J. Goodman, and A. Kägi (1996): Memory Bandwidth Limitations of Future Microprocessors, *International Symposium on Computer Architecture*.

[2] Y. Choi, A. Knies, L. Gerke, and T. Ngai (2001): The Impact of If-Conversion on Branch Prediction and Program Execution on the Intel Itanium Processor, *International Symposium on Microarchitecture*.

[3] N. Jouppi (1990): Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers, *International Symposium on Computer Architecture*.

[4] J. Hennessy and D. Patterson (1996): Computer Architecture: A Quantitative Approach, Morgan-Kaufman.

[5] J. Huang and D. J. Lilja (2003): Balancing Reuse Opportunities and Performance Gains with Sub-Block Value Reuse, *IEEE Transactions on Computers*, *52*, 1032–1050.

[6] M. Lipasti, C. Wilkerson, and J. Shen (1996): Value Locality and Load Value Prediction, *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[7] S. Mahlke, R. Hank, R. Bringmann, J. Gyllenhaal, D. Gallagher, and W. Hwu (1994): Characterizing the Impact of Predicated Execution on Branch Prediction, *International Symposium on Microarchitecture*.

[8] E. Rotenberg, S. Bennett, and J. Smith (1996): Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching, *International Symposium on Microarchitecture*.

[9] J. Shen and M. Lipasti (2003): Modern Processor Design, Fundamentals of Superscalar Processors, McGraw-Hill.

[10] J. Silc, B. Robic, and T. Ungerer (1999): Processor Architecture: From Dataflow to Superscalar and Beyond, Springer-Verlag.

[11] D. Sima, T. Fountain, and P. Kacsuk (1997): Advanced Computer Architectures, A Design Space Approach, Addison Wesley Longman.

[12] A. Sodani and G. Sohi (1997): Dynamic Instruction Reuse, *International Symposium on Computer Architecture*.

[13] J. Tsai, J. Huang, C. Amlo, D. Lilja, and P. Yew (1999): The Superthreaded Processor Architecture, *IEEE Transactions on Computers*, *48*(9).

[14] D. Tullsen, S. Eggers, and H. Levy (1995): Simultaneous Multithreading: Maximizing On-Chip Parallelism, *International Symposium on Computer Architecture*.

[15] S. VanderWiel and D. Lilja (2000): Data Prefetch Mechanisms, *ACM Computing Surveys*, *32*(2), 174–199.