Multithreaded Processors

Venkat Arun

Venkat Arun is a 3rd year BTech Computer Science student at IIT Guwahati. He is currently working on congestion control in computer networks.

In this article, we describe the constraints faced by modern computer designers due to operating speed mismatch between processors and memory units and some of the solutions adopted for overcoming them. We also give simple analytical models to give a basic understanding of the strengths and weaknesses of the various methods discussed.

1. Introduction

Gordon Moore, the cofounder of Intel^{TM}Corporation, predicted in a 1975 paper that the number of transistors that can be packed in one integrated circuit will double every two years [1]. The industry has faithfully followed this law. Since transistors are the fundamental building blocks of computers, the computing power accessible to us has been growing tremendously over the past few decades. Indeed, the processor present in a cheap modern mobile phone is more powerful than the computer aboard NASA's Apollo 11 mission to the moon.

Along with the increasing number of transistors per chip, similar improvements have been seen in the size of components, the power consumed and their speed of operation. Unfortunately, since 2004, the speed at which the transistors can operate has remained almost constant at one operation in every few fractions of a nanosecond. This means that although we can do more complex operations in a unit of time, the speed at which the operations can be done has not changed much. This has forced computer designers to look for alternate solutions where many operations are performed simultaneously, so that the number of operations performed increases

Keywords
Multithreading, processor parallelism, pipelined processors, processor architecture.

even though the time taken for each operation remains roughly the same. This can be prominently noted in the fact that the clock speeds of processors have stagnated at around 4 GHz for a very long time though their performance continues to improve.

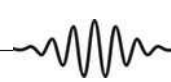
As processor speeds continue to increase, the gap between the speeds of RAM and processor continues to widen.

Another obstacle to faster computation is the fact that it takes very long to access data from the memory, (by memory access, we refer to writing and reading to and from the RAM), owing to the fact that it is very far away from the processor, where the data is actually manipulated. As processor speeds continue to increase, the gap between the speeds of RAM and processor continues to widen. That is, the processor can perform many operations in the time it takes for the memory access to complete after the request is issued. Hence, processors may waste time waiting for a memory operation and suffer a loss in performance.

To partly alleviate this problem, designers often place smaller and faster memories closer to the processor in the form of caches and registers. These contain the more-frequently-used data and thus reduce the average memory access delay (these delays are called latencies). Though a more detailed discussion of caches is out of the scope of this article, we note that despite these measures, memory access delays still remain a major concern, and will probably continue to be so [2]. We wish to reduce the adverse effects of this.

In this article, we discuss various ways to sidestep these obstacles. One way to do this is to have multiple streams of instructions (called threads) that can be executed (at least partially) independently of each other. There are two primary advantages of doing this:

1. We can have parallel circuits that perform several operations simultaneously, thus increasing the number of operations performed per second even



One method of alleviating the speed mismatch between a processor and RAM is to have multiple streams of instructions called threads that can be executed independently of each other.

though each operation takes a long time to complete.

2. Since we can switch between threads, this hides memory access delays (or latencies). That is, while a memory access is being performed for one thread, other threads can continue execution.

Another method to improve performance is to make the processor intelligent enough to perform multiple operations simultaneously, where all operations come from the same thread. We discuss many such methods one-by-one. Further, we also present simple mathematical models of each of these methods to compare their performance.

2. Pipelining

There are lots of ways to execute instructions in parallel without needing parallel streams of them from the programmer (a rather sophisticated and interesting method is described later, this section deals with a simpler method). We note that an instruction undergoes many steps during execution. It is first fetched from memory, then it is decoded and the relevant operands are fetched, after which it is executed and the result is written to the destination. Each of these steps has its own circuits. To improve performance, we treat these steps as stages of a pipeline through which instructions flow. Hence, each circuit executes a different instruction as shown in *Figure* 1. This is called pipelining of instructions.

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Figure 1. Pipelining of instructions.

Fetch Instruction	Read instruction operands	Perform operation	Write any results back

Table 1. An example showing pipeline steps.

Table 1 shows an example of the pipelining method. In this example, an instruction is: M[102] ← M[102] + M[34]. After fetching this instruction, the operands – M[102] and M[34] – are read and then are given to an adder to add. The result is written back to M[102].

However, if any circuit takes very long to complete its job, the rest of the circuits are forced to wait. For instance, if the operands are not in the cache and need to be fetched from memory, it can take a long time. During this time, the rest of the circuits have to wait, idling, while the fetching happens. This is called a pipeline stall.

Also, by default, instructions are executed one after another. But sometimes, this order is changed depending on circumstances. For instance, Program 1 (see p.852) executes the same set of instructions repeatedly. In this case, whether to execute instruction 6 or instruction 3 can be known only after executing instruction 5 (called a conditional branch/jump instruction). Hence, the instruction fetcher does not know which instruction to fetch next. This also causes the pipeline to stall.

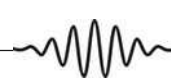
We now explore various methods to keep the circuits busy despite such delays. But before that, let's make a small analytical model of pipelined processors.

Let the total number of instructions be n out of which m instructions cause a stall. Further assume, simplistically, that the pipeline moves forward at every clock cycle unless there is a stall. If a stall takes time t' on average, then, time taken to execute n instructions is:

$$T = nt + mt' \qquad (1)$$

where t is one clock cycle time.

Pipeline stall occurs whenever there is delay in fetching operands or due to branch instructions.



In block-based multithreading, instructions of one thread are executed until it hits a long delay upon which it switches to another thread.

3. Multithreaded Cores

One way to keep the circuits busy is to have several streams of parallel and independent instructions called threads ready for execution. Pipelined processors can be extended quite naturally to handle this. Let's discuss some of the common methods:

3.1 *Block-Based Multithreading*

This technique simply continues executing instructions of one thread until it hits a long delay such as a branch or not finding data in the cache. Upon hitting such a delay, it switches to another thread. This thread also runs till it hits a long delay and the process repeats. Such a strategy enables us to *hide* long delays, but leaves out shorter delays where the cost of switching is greater than that of tolerating the delay.

Here, switching between threads also takes some time, say τ clock cycles. Out of a total of n instructions, let m_1 instructions have a small average[1] stall of $t'_1 < \tau$ clock cycles and let m_2 instructions have a large stall of $t'_2 > \tau$ clock cycles. For the sake of comparison, assume $m_1 + m_2 = m$ and $m_1 t'_1 + m_2 t'_2 = mt'$ so that the number of stalls and the average stall duration is the same as in the pipelining case. Then, time taken to execute n instructions is,

[1] Average weighted by the fraction of instructions of each type.

$$T = (n - m_1 - m_2)t + m_1 t'_1 + m_2 t. \qquad (2)$$

We use τ instead of t'_2 in the third term as the longer stalls are hidden by thread switching and instructions continue to execute. Hence, we have,

$$T = nt + m_1(t'_1 - t) + m_2(\tau - t). \qquad (3)$$

If $t'_1 < t'$ and $\tau < t'$, we have a considerable speedup as compared to the pipelining case (1). However, unlike in pipelining, the programmer has to specify several threads for this speedup to be effective. Also the circuitry is a little bit more complex as thread switching

has to be taken care of, wherein the program state is repeatedly stored and loaded.

In fine grained multithreading the processor executes one instruction of each thread one-by-one before moving back to execute an instruction of the first thread.

3.2 *Fine Grained Multithreading*

In this strategy, the processor executes one instruction of each thread one-by-one before moving back to execute an instruction of the first thread. It skips the execution of an instruction of any thread that is stalled (facing a long delay). This way, the pipeline is almost always full and the processor is kept busy. Such a processor needs a separate copy of registers and instruction counter for each thread, and hence, makes the processor more complex.

Note that such a system stalls only when all threads are stalled. Assume there are a total of n instructions out of which m cause a stall, and t' is the number of extra clock cycles required to complete a stalled execution. The probability that all p threads will be stalled at a particular clock is $\left(\frac{mt'}{nt+mt'}\right)^p$. Hence, if T is the number of clocks required to execute the n instructions, we have:

$$T = nt + \left(\frac{mt'}{nt+mt'}\right)^p T,$$

$$T = \frac{nt}{1-\left(\frac{mt'}{nt+mt'}\right)^p}.$$

This is because $\frac{mt'}{nt+mt'}$ denotes the expected number of clock cycles where the processor is stalled for a particular thread. Evidently, the execution time here is smaller than either of the two cases described above for a sufficiently large p.

Figure 2 summarizes the different pipelining methods.

3.3 *A Note of Caution*

The above formulae are approximate models and are not exact. For instance, when multiple threads access mem-

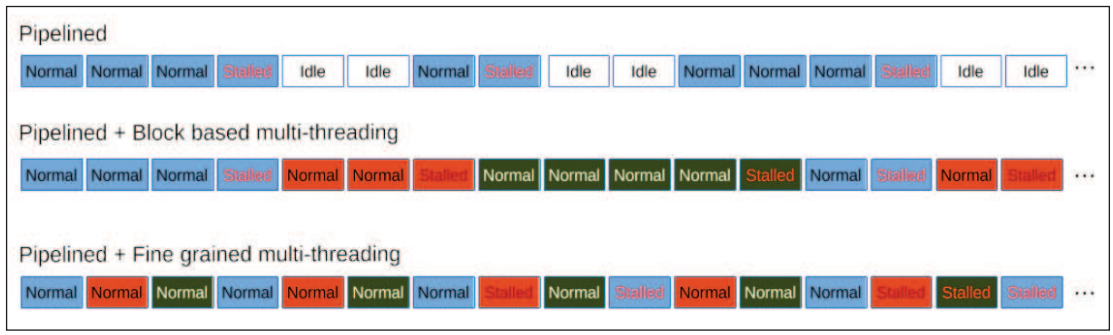


Figure 2. Various pipelining methods at a glance.

ory, its response time worsens, which we have not taken into account. Further, memory access time depends not only on the number of accesses but also on the access pattern (references [5] and [6] model this is detail). Also, a more complex processor is more expensive and likely to have a lower clock rate. Thus, executing more instructions per clock cycle does not imply executing more instructions per second. Hence, the speedup may not be as good as it looks in the formula. We have also assumed that all threads and all instructions have identical stall delays, which may not be true. There are many other assumptions not mentioned here. Hence, one must be cautious while using simple mathematical models.

4. Superscalar Processors

The methods we have looked at so far rely on the programmer to supply several streams of instructions that can execute at least partially independent of each other. Although this means that the resulting instruction streams are intelligently split up to ensure efficiency, it places a lot of burden on the programmer as automatic methods to parallelize serial programs have not yet been very successful. Further, lots of details are known only when the program runs. Hence, it may not be possible to decide beforehand, which operations are to be executed in parallel.

Thus, many modern processors are designed to automatically detect when instructions can be executed in parallel. That is, as long as the data to operate on and the circuit that performs the operation are ready, the processor executes the instruction even when all the preceding instructions have not yet been executed. Instructions are thus executed out of order. Such processors are called superscalar processors. Further, in case the processor encounters a conditional branch instruction (i.e., an instruction that asks the processor to change its line of execution of instructions based on a certain condition), the processor may not know which instruction to execute next before the condition is resolved and the address of the next instruction is calculated. Here, the processor simply guesses which instruction is likely to be executed next and continues execution. Of course, the processor must make note of the guess, so that in case it turns out to be wrong (after the address of the next instruction is known), the processor can undo the incorrectly performed operations. As you can probably imagine, the circuitry for doing this is rather complex and cleverly designed. We will discuss some of the elegant techniques employed to achieve this goal.

Many modern processors are designed to automatically detect when instructions can be executed in parallel.

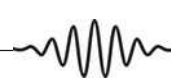
4.1 *Register Renaming*

Before we discuss the details of how superscalar processors work, let us look at an example program:

The following program finds the sum of numbers stored in memory locations 1 to 100 and saves the sum in memory location 102.

Here M[i] refers to the ith cell in the memory and $\leftarrow$ denotes copying of data from the right to the left.

In Program 1, each instruction accesses memory, which we already know takes a long time. Here, notice that M[101] and M[102] are used in every iteration which is slow and unnecessary. To handle such cases, most pro-



Box 1.

Program 1

1. M[101] ← 1
2. M[102] ← 0
3. M[102] ← M[102] + M[M[101]]
4. M[101] ← M[101] + 1
5. if M[101] ≤ 100, continue execution from instruction 3
6. ...

Program 2

1. A ← 1
2. B ← 0
3. C ← M[A]
4. B ← B + C
5. A ← A + 1
6. if A ≤ 100, execute instruction 3

cessors have a small number of very fast memory locations (typically 8, 16, 32 or 64 bits wide) called registers which are used to store commonly-used values. For instance, we could rewrite program 1 as follows (assume our processor has 5 registers named A, B, C, D and E):

With this system, we have removed 5/6th of the memory accesses which greatly speeds up execution. Further, it is clear that instructions 4 and 5 can be executed simultaneously if sufficient hardware is available. But we can do more than that. Note that here, reading from memory is the main bottleneck (instruction 3). Although it takes a long time to access memory, we can place accesses one after another, so that the number of accesses per second is quite high. In this case, we would like to place a lot of requests with the memory and add the values to B when the values arrive. The programmer only provides Program 2 and it is the cleverly designed hardware that performs these optimizations. Most processors do this by renaming *architectural registers* (as given in the program) A, ... , E to one of the *physical registers*, $R_1, R_2, ..., R_n$ (where n is fairly large, here let us take $n = 256$). The renaming is done to remove false dependencies between instructions. The following rule is used for the purpose (see Program 2):

```
i   A ← 1
ii  B ← 0
iii C ← M[A]
iv B ← B + C
v  A ← A + 1
vi if A <= 100,
execute instruction 3
vii C ← M[A]
viii B ← B + C
ix  A ← A + 1
x   if A <= 100,
execute instruction 3
xi  C ← M[A]
xii  B ← B + C
xiii A ← A + 1
xiv if A <= 100,
execute instruction 3
xv C ← M[A]
...
```

```
i     R1 ← 1
ii  R2 ← 0
iii  R3 ← M[R1]
iv  R4 ← R2 + R3
v  R5 ← R1 + 1
vi if R5 <= 100,
execute instruction 3
vii R6 ← M[R5]
viii R7 ← R4 + R6
ix  R8 ← R5 + 1
x   if R8 <= 100,
execute instruction3
xi  R9 ← M[R8]
xii R1 ← R7 + R6
xiii R2 ← R8 + 1
xiv If R2 <= 100,
execute instruction 3
...
```

Inst i	Inst ii	Initiate load 1 (Inst iii)	
Inst v	Inst vi, branch predict	Initiate load 2 (Inst vii)	
Inst ix	Inst x, branch predict	Initiate load 3 (Inst xi)	load 1 completed
Inst iv	inst xiii	Initiate load 4 (Inst xv)	load 2 completed
Inst vii	Inst xiv, branch predict	Inst viii	load 3 completed
Inst xii	...	...	...

Table 2.

Note: The rightmost column represents the instructions executed at each time slice. Notice that we are able to execute many instructions in a single cycle where a normal processor could take many cycles for a single instruction. Here, each row represents a time slice (clock cycle) and the entries represent operations which are performed in that time slice. Note that we execute instructions iii, v, vi, ... before instructions iv, vii, ... as the latter depend on the former and the former instructions do not depend on the latter and hence can be executed beforehand.

Whenever a normal register, X is being written to, rename it to a free register, R_1. After that, replace every occurrence of X with R_i until X is written to again, in which case, it is renamed to another free physical register. A physical register is deemed to be free if the last instruction to read from it has finished execution and has exited the processor. For instance, the processor could rename a run of Program 2 as shown in *Table* 2.

4.2 *Further Optimizations*

Sometimes a memory address that was just written to is accessed again. In such situations, many processors shortcircuit this process by bringing the value directly into execution before sending it to be written into memory. Further, if the processor was made to wait at each branch instruction (here instruction 6), the amount of parallelism that could be exploited decreases greatly. Hence, many processors guess which instruction to execute next and continue execution. In case the guess is

A multi-threaded superscalar processor is simply a superscalar processor that executes instructions from multiple threads, each having separate logical instruction streams, registers, etc., but sharing a majority of the physical resources.

wrong, they undo all incorrectly executed instructions.

4.3 *Superscalar Multithreading*

Much like pipelining, superscalar architecture also extends very naturally to support multiple threads of instructions. A multi-threaded superscalar processor is simply a superscalar processor that executes instructions from multiple threads, each having separate logical instruction streams, registers, etc., but sharing a majority of the physical resources. The additional circuitry required to support multiple threads is minimal but the performance improves greatly [3]. Commercially this is called Hyperthreading$^{\mathrm{TM}}$.

Due to the nature of superscalar processors, it is difficult to analyze them using 'time taken to execute n instructions'. Hence, we try to estimate their throughput (the number of instructions executed per cycle) as follows.

Let each instruction type in the processor be represented by I_j, and let the processor be capable of executing each instruction with a maximum throughput of $T_{h\max}(I_j)$. It represents the maximum number of instructions of type I_j that can be executed if there were no other delays in the processor. One can assume that there will be at least one instruction type (say, I_k) that is a bottleneck. That is, it is executing as fast as it can, $T_h(I_k) = Th_{\max}(I_k)$, where $T_h(I_j)$ is the actual throughput in the program. Let the program contain $P(I_j) * 100\%$ of instructions of type I_j for each j. Then,

$$T_{h\max}(I_k) = T_{h\mathrm{net}} * P(I_j)$$

Hence, we get the net throughput as $T_{h\mathrm{net}} = \frac{T_{h\max}}{P(I_j)}$. But, here we have not considered branch prediction errors. Let us assume that branch misprediction forces us to discard $e_{\mathrm{b}} * 100\%$ of the instructions, then we get the

effective throughput as,

$$T_{h\text{net}} = (1 - e_{\text{b}})\frac{T_{h\max}}{P(I_j)}$$

Note, however, that superscalar processors are optimized for executing instructions from a single thread and multithreading capability is added as an afterthought. The architectures based on pipelining, however, base their performance on having multiple threads.

Superscalar processors are optimized for executing instructions from a single thread and multithreading capability is added as an afterthought.

In the past decade, a new model of computing has come up in the form of GPUs which are highly multithreaded, often supporting thousands of very simple processing cores. [8] gives a nice overview of the different ways to exploit parallelism, including via GPUs.

Suggested Reading

[1] **Gordon E Moore, Progress in Digital Integrated Electronics, JEEE International Electron Devices Meeting, 1975, Technical Digest, pp.11–13.**

[2] **Wm A Wulf and Sally A McKee, Hitting the memory wall: implications of the obvious, *ACM SIGARCH Computer Architecture News*, 23.1, 20-24, 1995.**

[3] **Dean M Tullsen, Susan J. Eggers and Henry M Levy, Simultaneous multithreading: Maximizing on-chip parallelism, *ACM SIGARCH Computer Architecture News*, Vol.23, No.2, ACM, 1995.**

[4] **Susan J Eggers et al., Simultaneous multithreading: A platform for next-generation processors, *Micro, IEEE*, 17, 5, 12-19, 1997.**

[5] **Dhruba Chandra et al., Predicting interthread cache contention on a chip multiprocessor architecture, High Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on, *IEEE*, 2005.**

[6] **Xi E Chen and Tor M Aamodt, A first-order fine grained multithreaded throughput model, *High Performance Computer Architecture*, 2009. HPCA 2009. IEEE 15th International Symposium on IEEE, 2009.**

[7] **John L Hennessy and David A Patterson, Computer architecture: a quantitative approach, Elsevier, 2012.**

[8] **http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html**

Address for Correspondence
Venkat Arun
Barak Hostel
IIT, Guwwahati, Amingaon
Guwahati 781039
Email:
venkatarun95@gmail.com