

# Instruction-level Parallelism

Report for Software View of Processor Architectures

COMP9244

Godfrey van der Linden

2006-04-13

## 1. Introduction

When people make use of computers, they quickly consume all of the processing power available. It is the nature of computers; their flexibility quickly leads to ideas for more automation, which then lead to more and more ideas, until the processing resources available are exhausted. In addition software development is agglomerative, it is easier to add software and hence CPU load than it is to remove it. This is a fundamental asymmetry in software development. As we software developers have developed more sophisticated ideas for developing and layering our software; and users have demanded more of our software infrastructure, more demands have been made of the underlying hardware.

A very simple equation quantifies the performance of hardware.

$$\text{Instructions-per-second} = \text{instructions-per-cycle} \times \text{cycles-per-second}$$

Fortunately Moore's law has given us a formidable, non-linear, improvement in performance for some time. Hardware design has been incredibly competitive between teams of engineers in companies. For a long time the primary metric to measure performance was cycles-per-second (CPS), which caused the megahertz myth. Intel has done more to explore the boundaries of CPS in common CPUs than any other company. Customers of Intel found that another metric than CPU frequency had become increasingly important in their buying decisions however, the metric of instructions-per-second-per-watt.

In parallel with research into ever-increasing cycles-per-second, designers have also been improving the instructions-per-cycle (IPC) that their hardware is capable of. When they crossed the boundary of greater than one instruction-per-cycle ( $IPC > 1$ ) they entered the world of instruction-level parallelism (ILP). This paper describes the primary techniques used by hardware designers to achieve and exploit instruction-level parallelism.

This report is organised as follows: in the next section I shall quickly outline pipelining—which is required background to understanding any instruction-level parallelism implementation. Section 3 discusses the several hazards that instruction-level parallelism (ILP) uncovers and techniques for working around the hazard or otherwise minimising the impact of them; it also discusses extensions to ILP such as superscalar processors. Section 4 describes the theoretical limit of ILP attainable by ideal and potentially realisable processors. In the conclusion I discuss what impact these techniques have on operating system development.

In this document I shall be discussing RISC based instruction architectures. This is not especially constraining as it is possible to map the register-memory/memory-memory CISC instruction-set to a series of RISC partial instructions. This can be done mechanically and is used by Intel and IBM in the Pentium & Power processor families.

## 2. Background—pipelining

When a CPU executes an instruction, it transitions through number of stages to complete. These basic stages (or phases) are: instruction fetch, instruction decode, register fetch, execution/effective-address computation, memory access and write back to registers.

*Instruction fetch (IF)*—use the current program counter (PC) to index into memory and fetch the instruction. The act of fetching from memory may cause an exception and other complicated interactions with the hardware memory optimisations.

*Instruction decode/Register fetch (ID)*— this phase combines decoding of the next instruction with the latching of the current register values from the register file with the interpretation of the operation to be executed. A suitably designed instruction-set architecture it is possible to determine the registers operated on by the command independently of what the actual instruction is, which allows the register fetch to occur in parallel with the instruction decode.

*Execution/effective-address computation (EXE)*—the arithmetic logic unit (ALU) is central to instruction execution. Given the decoded operation and latched operands, the ALU performs the required operation, which can be either register arithmetic or an effective-address computation. The result is then latched for the next phase of execution.

*Memory access (MEM)*—either store a latched operand into memory indexed by the effective-address(EA) computed in the previous phase or load and latch a value from the computed EA.

*Write back to registers (WBR)*—Transfer the latched output of the memory access or execution phases to the required register. Some instructions do not need a WBR phase.

A simple CPU implementation would perform each phase in a clock cycle and every instruction would take up to 5 clock cycles to complete. Not all instructions require all phases, for instance a simple register indirect store instruction does not need to perform the execution or the write back phases and can complete in 3 cycles.

Early computer designers realised that this implementation leaves most of the CPU idle for most of the time and does not use the resources efficiently. In 1960 the designers of the IBM-7030 (Stretch—because it was supposed to be a stretch in the state of the art) introduced the concept of an instruction pipeline. As Figure 1 below shows, a pipeline can overlap the execution phases of an instruction with previous instructions.

Instruction\clock	1	2	3	4	5	6	7	8	9
i	IF	ID	EXE	MEM	WBR				
i+1		IF	ID	EXE	MEM	WBR			
i+2			IF	ID	EXE	MEM	WBR		
i+3				IF	ID	EXE	MEM	WBR	
i+4					IF	ID	EXE	MEM	WBR

Figure 1: Simplified instruction pipeline, with instruction overlap it is possible to complete one instruction per clock cycle.

### 2.1. Pipeline hazards

The simple pipeline presented in Figure 1 is very easy to implement. Consider the vertical bars between phases for the *i*-th instruction in the figure. In a simple multi-clock design these bars are latches that store the output of one phase across a clock cycle. To pipeline such a processor, all that is required is to use the latches insulate the execution of one instruction phase from the next instruction another. Hence the IF unit latches the output of the *i*-th instruction on cycle 1 and the next instruction in cycle 2. A wonderful, elegant, design that achieves a five times performance improvement in instruction throughput.

Unfortunately this solution is over-simple and will not work. When one stage of a pipeline depends on the results of an earlier still incomplete instruction, the later instruction will execute with wrong results. Technically this is a hazard is known as a read after write (RAW) *data hazard*. As execution is in order we will not address other possible hazards such as write after write or write after read until later.

When a branch needs to be taken then the next instruction cannot be determined—in this simple design—until the instruction pointer (IP) register WBR phase has been completed. This is known as a *control hazard*, though in this case it is a variant of a data hazard, as the pipeline continues executing a simple linear instruction stream for a brief period after the branch.

It is also possible that two instructions need to use the same hardware in the same cycle; this is called a *structural hazard*. For instance on hardware with a single memory interface then the MEM phase of a load instruction will interfere with the IF phase of a later instruction.

## 2.2. Solutions to pipeline hazards

### 2.2.1. Instruction pipeline stalling

An obvious solution for any of the hazard as described earlier is to stall the pipeline until the hazard clears. For instance, if an ID phase requires a register to be read and that register is being modified by an earlier instruction then stall the ID phase until the WBR phase of the earlier instruction completes. This slows a pipeline down and increases the cycles per instruction (CPI) but we have achieved some overlapping of instructions. As not all instructions would suffer this hazard we will lower the CPI on average.

Competitive pressure between hardware design teams led to the development of alternatives to stalling the pipeline for hazards, the goal being to have and maintain a CPI of one. Remember in the limiting case of too many pipeline stalls, the processor will degrade to a simple multi-clock cycle processor and with no significant overlap and a CPI of around five.

### 2.2.2. Duplicate units or functionality to avoid structural hazards

When the hazard is structural there is little that can be done to avoid it, no hardware can do two jobs at once. The only solution for structural hazards is to add more hardware or to duplicate it. For example the ‘Harvard architecture’ of separate code and data caches can be considered a duplication of memory caches. It clears the structural hazard between the IF phase and the MEM phase by dual porting access to main memory.

The simple pipeline above already duplicates some ALU functionality. A simple adder in the IF phase allows the fetcher to compute the next inline instruction without using the ALU. However this is going to cause a control hazard if the instruction being fetched by the IF phase follows a branch. Control hazards will be discussed in more detail in the branch prediction later.

### 2.2.3. Register forwarding around data hazards

A data hazard can often be bypassed by forwarding the result from one of the cross-phase latches directly to the input of a phase in a later instruction. Consider the following code fragment:

```
add r1, r2, r3
sub r4, r1, r5
```

A data hazard exists between the computation of `r1` in the `add` and the use of `r1` in the `sub` instruction following it. The problem is that the register file is not updated until the WBR phase some two clock cycles after the result was computed by the EXE phase of the `add` instruction. Using multiplexers it is possible to ‘forward’ the result of the `add` EXE phase into the next EXE phase for the subsequent `sub` instruction. This technique recognises, that although the value has not yet been committed to the register file, it has been computed and is available in a latch when an EXE phase completes.

Unfortunately not all data hazards can be resolved in this way. Consider the fragment

```
ld r1, [r2]
add r2, r1, r1
```

Remember, from Figure 1, that the memory phase of the  $i$ -th instruction and the execution phase of next instruction are on the same clock cycle. As the `ld`'s data will not be available until the end of the MEM phase's clock cycle but the `add` EXE phase needs it at the beginning of the same clock. We must stall the pipeline for this hazard.

## 2.2.4. Minimising control hazards

Any branch in the instruction stream for the simple pipeline will cause not only a stall but also an unnecessary fetch of the next instruction after the branch, which could trigger a memory exception that isn't real. For instance if the very last instruction of the last text page is a branch to the beginning of a loop then the fetch unit will still attempt to read the following instruction and could trip a page fault. One possible solution is to refetch the next instruction after a branch and ignore any earlier page faults thus turning the instruction effectively into a stall. If we take a stall for every branch that taken we would be introducing a 10–30% decrease in instruction throughput depending on the branch frequency.

There are a few different methods of dealing with control hazards and minimising the occurrence of branch stalls. They are more or less complex and are in any case superseded in modern processors with branch prediction, which will be described in a later section.

## 3. Instruction-level parallelism

Most processors since 1985<sup>1</sup> have used the pipeline method just described to overlap instructions. The background material discussed a simple pipeline and how to achieve instruction overlap, this section will discuss far more sophisticated approaches to processors. Achieving not instruction overlap but the actual execution of more than one instruction at a time through dynamic scheduling and how to maximise the throughput of a processor. It will be useful to re-visit the various dependencies and hazards again, before discussing these more powerful techniques for identifying and exploiting more ILP.

All of the subsequent techniques in this paper exploit parallelism among instructions. The amount of parallelism available within a single basic block is really quite small; for typical RISC processors, without extraordinary optimisation techniques by the compiler, the dynamic branch frequency is between 15% and 25%, which means that the average basic block is between 3 and 7 instructions long. These instructions usually depend on each other too and as a result the amount of overlap is further limited. The upshot of these observations is that some way must be found to exploit parallelism across multiple basic blocks.

In this section I shall discuss a number of techniques that the processor can use alone. The compiler can also be of assistance in generating larger basic blocks by various techniques, such as loop unrolling. The compiler techniques available are in common with the VLIW/EPIC techniques and will not be covered in this paper.

### 3.1. Dependencies and hazards

Determining how one instruction relates to another is critical to determining how much parallelism is available to exploit in an instruction stream. If two instructions are not dependent then they can execute simultaneously—assuming sufficient resources that is no structural hazards. Obviously, if one instruction depends on another, they must execute in order though they may still partially overlap. It is imperative then, to determine exactly how much and what kind of dependency exists between instructions. The following sections will describe the different kinds of non-structural dependency that can exist in an instruction stream.

---

<sup>1</sup> [Hennessy03] p172

There are three different types of dependencies: data dependencies (aka true dependencies), name dependencies and control dependencies.

### 3.1.1. Data dependencies

An instruction  $j$  can be considered data dependent on instruction  $i$  as follows: directly, where instruction  $i$  produces a result that may be used by instruction  $j$  or indirectly, where instruction  $j$  is data dependent on instruction  $k$  and  $k$  is data dependent on  $i$  etc. The indirect data dependence means that one instruction is dependent on another if there exists a chain of dependencies between them. This dependence chain can be as long as the entire program! If two instructions are data dependent, they cannot execute simultaneously nor be completely overlapped.

A data dependency can be overcome in two ways: maintaining the dependency but avoiding the hazard or eliminating a dependency by transforming the code. Code scheduling is the primary method used to avoid a hazard without altering the dependency. Scheduling can be done in hardware or by software—in this paper, in the interests of brevity, only the hardware-based solutions will be discussed. The VLIW/EPIC report will cover software-based code scheduling.

A data value may flow between instructions through registers or memory locations. When registers are used, detecting the dependence is reasonably straightforward as register names are encoded in the instruction stream. Dependencies that flow through memory locations are much more difficult to detect as the effective-address of the memory location needs to be computed and the EA cannot be determined during the ID phase. Compilers can be of great help in detecting and scheduling around these sorts of hazards; hardware can only resolve these dependencies with severe limitations.

### 3.1.2. Name Dependencies

The second type of dependence is a name dependency. A name dependency occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between them. There are two types of name dependencies between an instruction  $i$  that proceeds instruction  $j$ : an anti-dependence occurs when  $j$  writes a register/memory that  $i$  reads (the original value must be preserved until  $i$  can use it) or an output dependence occurs when  $i$  and  $j$  write to the same register/memory location (in this case instruction order must be preserved.)

Both anti-dependencies and output dependencies are name dependencies, as opposed to true data dependency, as there is no information flow between the two instructions. In fact these dependencies are a direct result of re-purposing registers, hence an instruction-set architecture with sufficient registers can minimise the number of name dependencies in an instruction stream. Since name dependencies are not truly dependent they can execute simultaneously or be reordered, if the name used in the instruction is changed so that they do not conflict. This technique is known as register renaming and uses a bank of additional temporary registers in addition to the register file.

### 3.1.3. Data Hazards

A data hazard is created whenever there is a data dependency between instructions and they are close enough to cause the pipeline to stall or some other reordering of instructions. Because of the dependency, we must preserve program order, that is, the order in which the instructions would execute in a non-pipelined sequential processor. A requirement of ILP must be to maintain the correctness of a program and reorder/overlap instructions only what correctness is not at risk.

There are three types of data hazards: read after write (RAW)— $j$  tries to read a source before  $i$  writes it—this is the most common type and is a true data dependence; write after write (WAW)— $j$  tries to write an operand before it is written by  $i$ —this corresponds to the output dependence; write after read (WAR)— $j$  tries to write a destination before  $i$  has read it—this corresponds to an anti-dependency. Self evidently the read after read (RAR) case is not a hazard.

### 3.1.4. Control Dependencies

The last type of dependency is a control dependency. A control dependency determines the order of an instruction  $i$  with respect to a branch, so that  $i$  is executed in correct program order only if it should be. The first basic block in a program is the only block without some control dependency. Consider the statements:

```
if (p1)
    S1
if (p2)
    S2
```

S1 is control dependent on p1 and S2 is control dependent on p2 but is not dependent on p1. In general there are two constraints imposed by control dependencies: an instruction that is control dependent on a branch cannot be moved *before* the branch and, conversely, an instruction that is not control dependent on a branch must not be moved *after* the branch in such a way that its execution would be controlled by the branch.

A control dependency does not in itself limit performance fundamentally. We can execute an instruction path speculatively, provided we guarantee that speculatively executed instructions do not effect the program state until branch result is determined. This implies that the instructions executed speculatively must not raise an exception or otherwise cause side effects.

## 3.2 Overcoming data hazards with dynamic scheduling

### 3.2.1. Scoreboarding

Simple pipelines fetch an instruction and issue it unless there is a hazard that can't be hidden or forwarded around as described earlier. Sometimes a compiler may choose to generate a stream of instructions in such a way as to avoid hazards, this is known as *compiler* or *static scheduling*. Many processors use another approach, that of *hardware* or *dynamic scheduling*. In 1964 Control Data introduced the CDC6600, which was unique in many ways. It was the first processor to make extensive use of multiple functional units, dynamic scheduling and out-of-order instruction execution.

All of the techniques described so far use, in order instruction issue, which means that when an instruction is stalled in the pipeline, no later instruction can proceed. However we would like, as far as possible, for instructions to be issued when they are ready, not necessarily in order, hence *out-of-order* execution. To implement out-of-order issue we need to split the instruction decode phase into two: *issue*—decode instructions and check for structural hazards; *read operands*—wait until no data hazards obtain then read the operands and start executing.

When dynamically scheduling a pipeline, all instructions pass through the issue phase in order; however they can stall or bypass each other in the read operands phase and enter, or even, complete out of order. The CDC6600 used a *scoreboard* to keep track of the instructions while they are waiting to start or as they complete their execution phase. The goal of a scoreboard is to maintain processor throughput of one instruction per clock cycle (when there are no structural hazards) by executing instructions as early as possible. Hence, if the next instruction would stall then store it on a queue and start with a later instruction. The scoreboard takes full responsibility for instruction issue and execution. To take advantage of out-of-order execution multiple functional units were needed to clear structural hazards, the CDC6600 had 16 separate functional units.

Every instruction goes through the scoreboard, where the data dependencies are recorded. The scoreboard then determines when the instruction can safely read its operands and begin execution. If the scoreboard decides the instruction cannot execute immediately, it monitors the results of all functional units until the dependencies are satisfied and then executes the delayed instruction. In the meantime it can issue other, non-dependent, instructions.

### 3.2.2. Tomasulo's solution for dynamic scheduling

Tomasulo introduced a natural extension to the scoreboard solution for the design of the IBM 360/91, which was introduced in 1966. Invented by Robert Tomasulo, the scheme tracks operands to avoid RAW hazards and introduces register renaming, to eliminate WAW and WAR hazards. Many modern processors use variations on this scheme now.

RAW hazards are avoided by executing instructions only when operands are available. While an instruction is waiting it is stored in a reservation station. Reservation stations keep track of pending instructions and are implemented as a little queue of instructions waiting for functional unit and operand availability. Register renaming eliminates WAR and WAW hazards—which arise from name dependencies. This allows a processor to use a much larger register file than the instruction-set architecture can describe, for instance the Power4 has 32 general purpose registers in its instruction set but internally it has some 80 physical registers.

Tomasulo architecture executes instructions in three phases; each phase may take more than one clock cycle:

Phase 1—*issue*—gets the next instruction from the fetch unit. If there is a reservation station available then insert the instruction with the currently ready operands, if there is no matching station then a structural hazard exists and the instruction stream is stalled until a station becomes available.

Phase 2—*execute*—each reservation station monitors the output of the functional units and as each operand becomes available the station stores it. When all operands are available, the appropriate functional unit can execute the instruction. By delaying instructions in this phase we eliminate RAW hazards. To preserve exception behaviour, no instruction is allowed to initiate execution until all preceding branches are finalised. This guarantees that an instruction causing an exception really would be executed but at a cost. This cost can, mostly, be avoided by using speculation.

Phase 3—*write results*—when the execution finishes and the results are available, write them onto the internal common bus and route them into both the register file and any reservation stations that depend on the result. A memory store is usually saved in a separate queue until both address and data are available and then they are sent to the memory unit.

### 3.2.2. Control dependencies—branch prediction

Having dealt with data dependencies using Tomasulo's approach, we still need to deal with the other source of pipeline stalls, that of control dependencies. As we achieve more ILP it becomes vital to find a solution to control dependency stalls. As noted earlier, a basic block tends to be from 3–7 instructions long, if we can only achieve as little as three *instructions per cycle* (IPC) we would need to taking a pipeline stall either every cycle or every other cycle! Branch prediction is used to guess the outcome of a branch before we know for certain and is usually performed by the instruction fetch unit so that it doesn't have to stall the pipeline.

In modern architectures branch prediction is quite complicated and consists of a pair of individual predictors that are selected between. A predictor is a simple saturating  $n$ -bit counter, such that  $0 - 1 = 0$  and maximum value  $(2^n - 1) + 1 = \text{maximum value}$ . Each time a particular branch is taken its entry is incremented otherwise it is decremented. If the most significant bit in the counter is set then predict that the branch is taken. Although the general  $n$ -bit solution is given here, it has been shown that a simple 2-bit counter does nearly as well. For local branch prediction, the entries index is generated from the least significant bits of the address of the instruction after the branch.

A global branch predictor tracks the result of the last  $n$  branches in a  $n$ -bit shift register, that is if a branch is taken then a 1 is shifted in otherwise a 0 is entered. The shift register value is used as an index into a table of  $n$ -bit predictors. A final level of branch prediction is often used, that of the tournament predictor. The tournament predictor is a table of predictor entries which tracks correctness of global or local predictors, but is otherwise a standard  $n$ -bit predictor. The tournament table is often indexed using the local branch predictor method, though the Power4 xors the lower addressing bits with the global predictor's shift register value.

In this paper I shall not describe the history and development of various branch prediction strategies, rather I shall describe the strategy used by the Alpha 21264. The 21264 used the most sophisticated branch predictor available in 2001 and it is still a very good predictor. The branch predictor is implemented as a tournament between global and local predictors. The global predictor is implemented as a shift register of the last twelve

branches; this 12-bit number is used to index a 4K-entry table of 2-bit predictors. The local predictor has two levels: the top level is a 1024 entry local history table, indexed by 10-bits of the branch address<sup>2</sup>. Each entry consists of the outcome of the 10 most recent branches. This 10-bit history allows patterns of up to 10 branches to be discovered and predicted. The history is used to select an entry in another table of 1024 3-bit saturating counters. Finally the tournament between global and local predictors uses a table of 4096 2-bit predictors indexed by the branch address. For the SPECfp95 benchmarks the predictions were only wrong less than once per thousand completed instructions, the SPECint95 was slightly worse at 11.5 times per 1000 completed instructions.

### 3.2.3. Even more ILP using multiple issue/superscalar processors

Even with dynamic scheduling and branch prediction, instructions can stall in the issue phase if there are no reservation stations available for the instruction. Having dealt with data and control dependencies, hardware designers turned to the addition of additional functional units to clear structural hazards. An obvious, in hindsight, extension to Tomasulo's algorithm is to create multiple reservation station queues, one queue for each functional unit. A dynamically scheduled processor that can support multiple issue to a number of functional units is called *superscalar*.

This causes additional complexity though, as each functional unit probably runs in a different number of clock cycles and has different pipeline characteristics. For instance an FP add can be done in a two cycles and a FP multiply a few cycles more, both the add and multiply units would be pipelined allowing multiple instructions to overlap. However, a traditional divide unit can take tens of cycles and almost certainly cannot be pipelined.

However, multiple units and multiple threads of execution within an instruction stream makes the delivery of precise exceptions extremely difficult. Many hardware specifications require exceptions to be delivered at precisely the instruction that caused the exception, as the software developed for earlier iterations of the hardware's family came to rely on it. This is impossible when a later instruction has completed out-of-order on one of the other functional units. Speculation and in order committing of instructions avoids this problem on multiple issue processors, see below.

Once multiple instructions per cycle can be executed, branch prediction is no longer sufficient to keep all of the functional units in a superscalar CPU busy. Designers also needed a high-speed instruction fetch unit and wider pipes to memory. This leads to the promotion of IF phase section into to a sophisticated functional unit in its own right with complex branch and target prediction hardware.

### 3.2.4. Speculation

The use of superscalar techniques in combination with typical branch frequencies implies that a processor will execute a branch every clock cycle to maintain efficient use of hardware resources. We must overcome the limitations imposed by control dependencies to exploit the superscalar hardware efficiently. With the accuracy achieved by modern branch prediction units, hardware now can execute instructions speculatively, with a near certainty that this speculation will be used. This technique involves executing post branch instructions as if the prediction is correct and unwinding any speculation if it later turns out that prediction was wrong. The higher the accuracy of the branch predictors, the more leeway designers have in the number of cycles it take to get the instruction stream back on track after a prediction failure.

The hardware to support speculative execution is, yet another, extension to Tomasulo's basic design. The extension separates the flow of results between instructions from the completion and writing back of those results. While a series of instructions are still speculative, results are passed directly from instruction to instruction and the result is only stored in the register file when the instruction is no longer speculative. This additional phase is called the *instruction commit* phase.

---

<sup>2</sup> These bits are the least significant bits of the branch address /  $\text{Size}_{\text{inst}} + 1$ .

The key ideas are to allow instructions to execute out of order but to force them to commit *in order* and also to prevent any irrevocable action until they commit. Adding this commit phase to Tomasulo's approach uses a set of hardware buffers to temporarily hold the results of finished but not yet committed instructions. These buffers are called *reorder buffers* (ROB). They are also used to pass intermediate results among speculative instructions as they execute. The updated 4 phases in instruction execution follows:

- Phase 1—*issue*—get an instruction from the instruction queue. Issue it if there is a reservation station available *and* there is a spare slot in the ROB. Cache the currently available operands from either the register file or earlier ROB entries into the reservation station.
- Phase 2—*execute*—wait until all of the operands are available for one of the reservation stations, then execute the operation. Instructions may use multiple clock cycles in this stage, hence may complete in any order.
- Phase 3—*write results*—when the result is available, write it on the bus tagged with the ROB index of the destination register, the reservation stations that are monitoring the bus for any operands so tagged will snap up the operand and store it for execution.
- Phase 4—*commit*—there are two possible actions to be taken at commit, depending on whether the instruction is a branch or not. The commit phase takes the instruction from the head of the ROB queue and if it is a regular instruction then the result is stored in either the register file or memory. Alternatively if the instruction is a branch we must know by now if it was predicted correctly or not. If the branch was predicted correctly then no further action need be taken otherwise when prediction fails then the entire ROB queue is flushed and the IF unit is re-started at the new IP.

The use of reorder buffers and in order commits vastly simplifies the generation of precise exceptions. While an instruction is speculative any exception is temporarily stored in the ROB but when the instruction reaches the head of the ROB queue, the processor can then raise the exception, flush the ROB and start the IF unit at the exception handler.

The biggest advantage of speculation is its ability to uncover events that would otherwise stall the pipeline early, such as cache misses. This advantage can work against us when the operation that is being speculated implicitly takes a long time, for instance a costly exceptional event. To maintain as much of the advantage as possible and minimise the disadvantage, most processors will only speculate about low cost exceptions such as first-level cache misses. When a potentially expensive exception occurs, such as a high level cache or TLB miss the processor will stall until the instruction causing the event is no longer speculative.

## 4. Limitations of ILP

To evaluate the maximum potential instruction-level parallelism possible, an instruction stream needs to be run on an ideal processor with no significant limitations. The ideal processor always predicts branches correctly, has no structural hazards and has an infinite number of ROB buffers and renaming registers. This eliminates all control and name dependencies, leaving only true data dependencies. This model means that any instruction can be scheduled on the cycle immediately following the execution of the predecessor on which it depends. It further implies that it is possible for the last dynamically executed instruction in the program to be scheduled on the first cycle.

To measure the available ILP, a set of programs—in this case the SPEC benchmarks—where compiled and optimised with a standard optimising compiler. The programs were then instrumented and executed to produce a trace of the instruction and data references. Every instruction in the trace is then scheduled as early as possible given the assumptions of perfect branch prediction and no hardware limits. Table 1 shows the average amount of parallelism available for 6 of the SPEC92 benchmarks. Three of these benchmarks are FP intensive and the other three are integer programs.

SPEC	Instructions issued
gcc	55
espresso	63
li	18
fpppp	75
doduc	119
tomcatv	150

Table 1: Table of SPEC benchmark and average instructions issued for an ideal processor.

Obviously the ideal processor is not realisable; a realisable but ambitious processor can achieve much less than the ideal processor. Table 2 explores a processor which can issue 64 instructions per clock with no issue restrictions, a tournament branch predictor of substantial size, perfect disambiguation of memory reference, 64 additional renaming registers for both integer and FP.

SPEC\Window	Infinite	256	128	64	32	16	8	4
gcc	10	10	10	9	8	6	4	3
espresso	15	15	13	10	8	6	4	2
li	12	12	11	11	9	6	4	3
fpppp	52	47	35	22	14	8	5	3
doduc	17	16	15	12	9	7	4	3
tomcatv	56	45	34	22	14	9	6	3

Table 2: Average instruction issue of SPEC Benchmark versus window size for instruction decode

The most startling observation is that with *realistic* processor constraints listed above, the effect of the window size for the integer programs is not as severe as for FP programs, this points to the key difference between these two types of programs, it seems that most FP programs are vector-based and can use loop-level parallelism to a much greater extent than the typical integer program.

## 5. Conclusion

Li et al in [Li03] discovered, when instrumenting power consumed by particular operating system routines, that instruction-per-cycle statistics could be used to predict the power consumed by a particular routine, i.e. IPC and power consumption are highly correlated. They explored this correlation a little using a power accurate simulator of the MIPS R10000 and found that some 50% of the power used by a routine could be directly attributed to the data-path and pipeline structures in a processor. These structures are used to control the ILP being exploited by the CPU. Unfortunately this interesting correlation was not explored further in their paper.

To make maximum use of the superscalar structures of a CPU is quite complicated and a good compiler is essential. One illustrative example can be found in [Gray05] which found that the gcc Itanium code generation was not very good. Using the Itanium performance tools, such as *perfmom*, they hand optimised a particularly important portion of the IPC path. The basic development iteration was to measure the stalls in the processor pipeline, develop a different instruction sequence to avoid the stalls, assemble, test and start to measuring again. In this way they reduced the cycles used by the portion of the IPC path from 508 cycles to an, astounding, 36 cycles. Although the Itanium is not a traditional RISC superscalar processor, many of the techniques used to order the operations on Itanium are directly applicable to superscalar processors through a combination of static and dynamic scheduling.

Interestingly hardware designers also use the fundamentally iterative technique of: measure, design, build, test and measure again; as a basic performance improving process too. They try to find a typical load for the CPU, called *benchmarks*, measure stalls and other problems and design a solution to the difficulties uncovered, build and test it. This leads to a cycle of ever improving benchmarks; unfortunately what hasn't been demonstrated is that benchmarks improvements lead to improvements in real production software. Everything depends on how representative of typical workloads the benchmarks are.

As operating system developers we can study the benchmarks used by hardware designers and classify them in terms of how representative they are of a typical OS implementation. I would suggest using the 'li' lisp based intSPECmark is the starting point. The nature of operating systems is to use tables of pointers to functions, to implement OS services. A classic example of this would be Unix's CDEVSW table that maps system calls to particular drivers as a table of function pointer vectors. Lisp is essentially implemented in this way with tables of function pointers linking objects together. Unfortunately Tables 1 and 2, indicate that modern processors cannot exploit much ILP on these types of workload. It would be interesting to instrument an OS and see how many pipeline stalls are generated in normal, typical, operation. Perhaps Apple's Shark performance tool would be appropriate.

Current research in microprocessor design seems to have reached the limit of ILP exploitation for a single thread. This has led to exploration of thread-level parallelism (TLP). TLP is easier to exploit using multiple processors, though now designers are also taking advantage of idle superscalar hardware by using symmetric multi-threading. As the theoretical study demonstrated it is very difficult to uncover significant ILP in any given instruction stream. It is much easier to execute two independent instruction streams and as they are independent they cannot suffer from data and control hazards with respect to each other.

In practice it seems that superscaling a processor beyond a certain amount leads to decreasing performance per watt. In today's microprocessor marketplace, the laptop is the highest margin computer being sold by manufacturers. Since a laptop can run from batteries the power consumed by CPUs has become a major system integration issue. Superscalar ILP increases the base power draw of a CPU by introducing vast number of gates to support it, this is a very active field of research now among the processor manufactures and seems to be emerging as the new metric against which a processor is judged. This bodes well for future performance versus power consumption of upcoming processors.

## A. References

[Gray05] Gray, Chapman, Chubb, Mosberger—Tang, Heiser (2005). *Itanium—A System Implementor's Tale*. In Proceedings 2005 USENIX Annual Technical Conference (Anaheim, CA, USA April, 2005)

[Hennessy03] Hennessy and Patterson (2003). *Computer Architecture A Quantitative Approach 3rd Edition*, (Morgan Kaufmann, San Francisco, CA, USA, 2003)

[Li03] Tao Li, Lizy Kurian John (2003). *Runtime Modeling And Estimation Of Operating System Power Consumption*, SIGMETRICS'03 June 10–14, 2003, San Diego, Ca, USA, 2003