Survey of Techniques for Exploiting Instruction Level Parallelism

Michael McKeown, Sam Payne
Department of Electrical Engineering, Princeton University
{mmckeown,spayne}@princeton.edu

***Abstract*—In order to to keep pace with performance trends, computer architects have utilized the increasing availability of transistors to implement creative techniques for exploiting ILP in the microarchitecture. These techniques have been key factors in increasing sequential processor performance. In this work, we survey many of these designs, implement them, and simulate them on a set of benchmarks to compare the performance. The goal is to understand the trade-offs and performance benefits associated with each technique in addition to learning about the implementation.**

Almost all of these techniques attempt to avoid stalls and enable further ILP. A processor that stalls on data hazards performs the worst, except in programs with low amounts of data dependencies. Allowing bypassing provides dramatic improvements. Further improvements are achieved by adding OoO issue, as this allows younger instructions to bypass older stalled instructions, reducing the number of total stalls. However, stalls introduced by OoO issue should not outweigh the stalls eliminated, as we experience serious performance degradation in this case. When stalls from OoO issue are avoided, dramatic performance increases can be observed. Lastly, we observed that single-fetch is a rather serious performance bottleneck in OoO issue processors, as the bottleneck lies in the ability to fill the instruction window. A superscalar processor with dual-fetch easily out performance an OoO issue processor with single-fetch.

1. Introduction

Over the past few decades, the increasing speed and availability of transistors, attributed to Dennard scaling [1], has led computer architects to consider creative ways to utilize the growing number of resources in order to keep pace with Moore's law [2], which suggests that processor performance will double every two years due to the ability to pack more transistors on a single die. While increasing clock frequencies have contributed their share, innovative architectural designs have been a key factor in boosting processor performance.

Many architectural techniques seeking to achieve higher performance attempt to take advantage of instruction level parallelism (ILP), the ability to execute instructions in parallel. One of the most primitive forms of this is instruction pipelining, where instructions are split into multiple stages allowing them to execute concurrently in different stages at the same time. For example, while one instruction is being fetched from memory, another can be decoded, yet another can be executed, and so on.

As a result of pipelining, read after write (RAW) data dependencies between instructions pose a problem, as the most up-to-date version of an operand needed by an instruction could be the result of an instruction in flight. One solution to this problem is to stall the instruction reading the operand until the in-flight instruction has completed, however, this limits ILP by preventing instructions from executing concurrently. Thus, a superior approach is to bypass the operand out of the instruction in-flight when the result is calculated and before it is written back to the register file. For instance, an instruction needing the value of an operand that is in-flight in the arithmetic logic unit (ALU) could bypass the value from the output of the ALU instead of waiting to receive it from the register file.

Superscalar processors utilize a more advanced technique for exploiting ILP. The processor consists of multiple pipes with separate functional units. The pipes may have different latencies. On each clock cycle, multiple instructions are issued, one down each pipe if possible. Thus, multiple instructions can be executing concurrently across pipes and on each pipe (as each pipe is pipelined as discussed above). As an example, there may be one pipe for the ALU, one pipe for memory accesses, and one pipe for the multiple/divide (muldiv) unit. On a given cycle an arithmetic or logic instruction, a memory instruction, and a multiply or divide instruction may be issued given the instructions are available and the operands are not pending.

Another advanced technique for exploiting ILP is executing instructions out of order (OoO). Often an instruction is not able to be issued due to data dependencies, that is, one or more of its operands are being written to by an older instruction which has not finished calculating it's result. Thus, the instruction must stall until its operands are ready. This occurs even when bypassing is present due to long latency instructions such as multiply or divide. In order to avoid stalling, OoO processors queue up fetched instructions and therefore have a pool of instructions from which to issue, often called the instruction window. Thus, if one instruction must stall due to a data dependency there are other instructions which can potentially be issued. OoO execution offers more potential for ILP by providing increased availability of instructions to issue.

OoO execution avoids stalling in the case of true data dependencies, a RAW hazard, however does not solve the problem of stalling in the case of an anti-dependence, a write after read (WAR) hazard, or an output dependence, a write after write hazard (WAW). In order to expose further ILP, stalling in these cases must be eliminated. This is accomplished through a technique called register renaming, where destination architectural registers are renamed to a different physical location, where the result is buffered until the instruction commits and the result is written to the architectural location. Thus, the ordering of writes to the architectural registers is preserved and correct operation is guaranteed while avoiding stalling in the case of WAR and WAW hazards.

Another approach to exploiting ILP are very long instruction word (VLIW) processors. VLIW processors allow for multiple instructions to explicitly be encoded into a bundle that executes concurrently. A bundle directly exposes the hardware architecture to software by making each pipe visible as an instruction slot inside a bundle. This pushes much of the complexity in the previous approaches to the compiler, which chooses which instructions can be placed in a bundle together. Thus, ILP can be achieved in the hardware while the scheduling of instructions that execute concurrently is done in software. For example, a bundle may consist of two slots for ALU instructions, two slots for memory instructions, and a slot for multiply or divide instructions.

In this paper, we provide a survey of these techniques and how they affect the performance on benchmarks with different characteristics. We implement and compare the performance of a 5-stage pipelined processor, a 5-stage pipelined processor with bypassing, a 5-stage two-wide superscalar processor with dual-fetch, a 6-stage OoO processor with in-order single issue and OoO execution, a 7-stage OoO processor with OoO single issue and a single in-flight memory instruction, a 7-stage OoO processor with OoO single issue and multiple in-flight memory instructions, and a 7-stage OoO processor with OoO multi-issue and multiple in-flight memory instructions. We do not consider VLIW in our survey, but present it above as an alternative to the designs we consider here.

The rest of this paper is structured as follows. Section 2 discusses related work and processors that utilize some of the discussed techniques. In section 3 we discuss the design of the processors implemented in this work and trade-offs that were made, and in section 4 we provide implementation details. In section 5 we articulate our experimental methodology followed by a presentation and discussion of our results in section 6. Section 7 concludes.

2. Related Work

Instruction level parallelism in the form of pipelining has been used for decades. A simple form of pipelining was first used in the Z1 and Z3 computers [3]. It was later used in the more traditional sense in the ILLIAC II [4] and the IBM 7030 Stretch [5]. Since 1970, almost all processors built have been pipelined. Today, pipelining is used in virtually ever processor.

The CDC 6600 [6] was one of the first superscalar processors, where multiple parallel execution units were available to execute different instructions concurrently. Though the machine was only capable of sustained execution of one instruction per cycle, it's microarchitecture is very similar to the superscalar machines of today. Interestingly, it was also one of the first computers to implement a scoreboard, a structure to keep track of where operands were present for bypassing at a given time. The first superscalar processors that were manufactured on a single chip were the Intel i960CA [7] and AMD Am29000 series and the first x86 superscalar processor was the P5 Pentium [8].

The concept of OoO processors was first articulated by Tomasulo in his algorithm that supported out-of-order issue and execution [9]. It buffered instructions in reservation stations, one for each execution unit. Instructions were checked for whether operands were ready and functional units were free and issued if so. This algorithm also described a method for register renaming, where destinations were assigned virtual registers which could be bypassed out of. The IBM 360/91 [10] was the first machine to implement a dynamic instruction issuing algorithm using Tomasulo's algorithm. The CDC 6600 [6], three years earlier, could be considered the first processor to perform out of order execution, as it issued instructions in-order but executed them out of order, however, it possessed no structure similar to register renaming and could not issue instructions out of order.

For many years, processors continued to issue one instruction per cycle. ILP utilization was pursued in different ways including vector processing utilized by the STAR-100 [11] and CRAY-1 [12] machines in the 1970's. These vector machines were some of the first to be called "supercomputers." Multiprocessing was also pursued as a method of extracting parallelism in computing as seen in machines such as the CRAY X-MP [13], appearing in the 1980's.

It was not until the 1980s that multi-issue processors appeared, introducing true superscalar processing. Some of the first commercial machines to include multi-issue include the IBM 6000 [14], and the Astronautics ZS-1 [15]. Processors allowing for multi-issue allow for more than one instruction to execute per cycle, producing an IPC count greater than one. The multi-issue superscalar design still serves as a base for most high performance processors today.

VLIW machine architectures also emerged in the 1980's. These machines move the ILP utilization responsibility to the compiler by forcing multiple operations into a single instruction. These instructions are issued as a unit, similar to issuing multiple individual instructions. In this case, the compiler decides what should be issued concurrently, not a dynamic issuing structure. The first of these machines included the ELI-512 [16]. This microarchitectural style continues to be built in modern processors, though it has not been as widely adopted in recent years. The Intel Itanium processor [17] serves as an excellent example of a modern VLIW processor.

3. Design

This section describes the design differences between each iteration of the processor implemented in this work. The 5-stage pipelined processor, 5-stage pipelined processor with bypassing, the 5-stage two-wide superscalar processor with dual-fetch, and 6-stage OoO processor with in-order single issue have all been described in previous lab reports written this semester in ELE475, so only a brief description is provided here. For OoO all processors implemented in this report, we used the Tomasulo Algorithm for register renaming [9]. This section describes the remaining designs, their function and all crucial decisions we made.

A. *5-Stage Pipelined Processor*

The 5-Stage pipelined processor is a simple processor with 5-stages: fetch, decode, execute, and writeback. No bypassing is present, all RAW data hazards require stalls.

B. *5-Stage Pipelined Processor with Bypassing*

This processor adds on bypassing hardware to avoid stalling in the case of RAW data hazards. All values are bypassed to the decode stage. Any functional unit and stage can be bypassed from as long as the data is ready at that stage.

C. *5-Stage Two-Wide Superscalar Processor with Dual-fetch*

The 5-stage two-wide pipelined processor with bypassing is augmented with dual fetch and multiple ALU functional unit pipes. In addition, the processor is capable of issuing two instruction simultaneously if possible. Steering logic is implemented to steer instructions down the correct pipe and take care of the case where one instruction must stall but not the other. In many cases, this allows for two instructions to be issued and executed concurrently and has dramatic potential for performance improvement.

D. *6-Stage OoO Processor with In-Order Single Issue*

Only OoO execution is added in this iteration of the processor, instructions are still issued in order from the decode stage. Since functional units have different latencies, instructions can arrive at the writeback stage OoO. Thus, a reorder buffer (ROB) and commit stage must be added in order to ensure instructions commit in order. Writeback hazards can also occur in this processor since two instructions are capable of reaching writeback at the same time. For example, a memory instruction (2 cycle latency) followed by an ALU instruction (1 cycle latency) will have a structural hazard on the writeback stage and the ALU instruction must stall.

E. *7-Stage OoO Processor with OoO Single Issue and Single In-Flight Memory Instructions*

To allow for OoO issue, two structures must be added to the 6-stage OoO processor: an IQ, and a rename table. An IQ holds instructions from using execution units until the data required for their execution is available while allowing ready instructions to use these execution units. The addition of the IQ requires an issue stage placed between the decode and execute stages in order to prevent an unreasonable cycle time. The rename table maps ROB slots to architectural registers. This renaming is necessary for preventing RAW and WAW hazards while instructions are issued out of order.

Two other structures, the scoreboard and ROB, must also be adjusted. The scoreboard tracks when data becomes available to instructions waiting to issue. Therefore, the scoreboard must communicate with the IQ to notify which instructions can be released. To simplify the design and to minimize

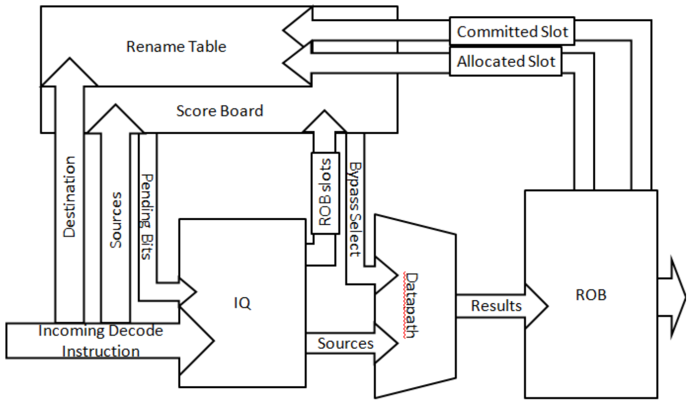

Fig. 1: High level communication of structures as instructions flow down the OoO pipe

communication signals, we built the rename table into the scoreboard.

Figure 1 shows the high level communication of structures as instructions flow down the pipe. Upon entering the issue stage, instructions send their sources to the scoreboard which notifies the IQ if the sources are pending. If the incoming instruction writes, the ROB allocates a slot as a destination. The destination of the incoming instruction is sent to the scoreboard and is paired in the rename table with the allocated ROB slot.

As data becomes available for bypassing, the scoreboard notifies the IQ. As ROB slots are committed, instructions in the IQ store the committed data. When all data needed by an instruction is available for bypassing or is present in the register file, that instruction can issue. As an instruction issues, the sources which require bypassing are sent to the scoreboard which determines the location from which the sources should be bypassed (the writeback stage or an ROB slot).

This design suffers from a number of drawbacks. Register renaming handles WAR and WAW hazards when writing to the register file, however this architecture includes no structure to prevent these same hazards from occurring in memory. Therefore, we allow only one memory instruction to enter the IQ to prevent memory instructions from passing one another. This design also only allows a single in flight branch since there is only one speculative bit available in the IQ - this will be detailed more in the Section 4.

F. *7-Stage OoO Processor With OoO Single Issue and Multiple In-Flight Memory Instructions*

Since allowing only one in-flight memory instruction limits performance, this architecture attempts to combat this problem. We chose to add a Load Store Queue (LSQ) to prevent WAR and WAW hazards in memory. The LSQ prevents memory instructions from accessing memory until the addresses of previous memory operations are known. In this way, memory address conflicts can be avoided when they arise to prevent memory aliasing. This requires separate adders for calculating addresses for loads and stores, which was previously coupled

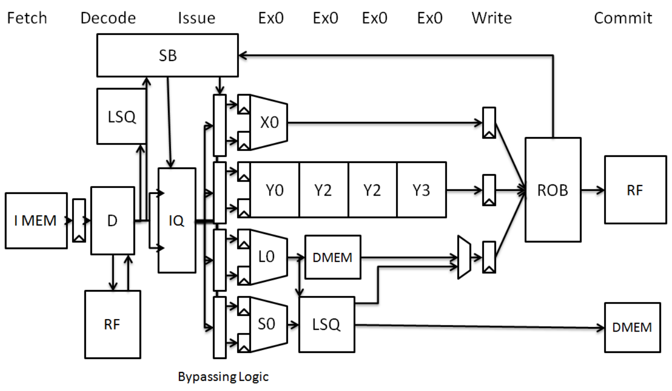

Fig. 2: Multiple-pipe architecture with separate adders for loads and stores and multiple writeback ports

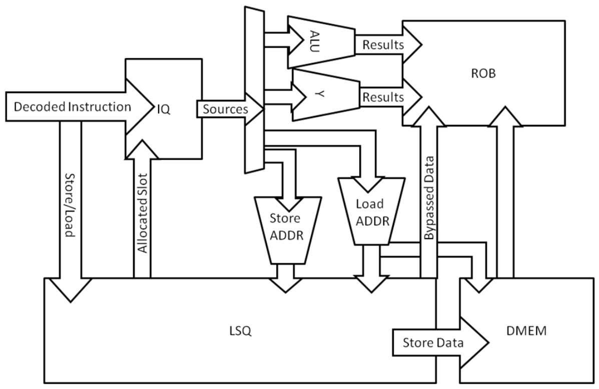

Fig. 3: High level communication of structures as instructions flow down the OoO pipe with the addition of a LSQ

with the ALU in the execute stage. In addition multiple writeback stages are added for each of the pipes as discussed below. The updated architectural diagram is shown in Figure 2.

Figure 3 shows the high level communication between the LSQ, IQ, and ROB. Upon entering the IQ, memory instructions are assigned a slot in the LSQ. There is no change in how memory instructions are issued. In execution, stores calculate their address and store this information in the LSQ. The LSQ chooses to commit stores when all previous stores and loads have completed.

Loads calculate their address and check to ensure that all the addresses of previous store instructions have been calculated and do not conflict. If a store's address has not been calculated, then the load will stall (address calculations have dedicated adders, so there is no structural hazard here). If a previous store has already calculated its address and it matches the address of the load, then the load will bypass the data which that store intends to write. Otherwise, the load will receive its data from memory. During load execution, the LSQ and memory are read in parallel and the result is multiplexed based on the signals returned from the LSQ.

The LSQ schedules stores to use the single ported data memory when loads are not using it. This does not cause any deadlocks since the store's data can be bypassed. If the data cannot be bypassed, the requesting load will stall to allow that store to use memory.

Since addresses are byte addresses, and load and store instructions can store whole words, the LSQ must be careful to check for address overlap, not just matches in addresses.

This design does however possess a fatal flaw which we were unable to solve due to time constraints. There is a possible deadlock scenario; consider the set of instructions in Listing 1. If the second load issues before the first, it will stall waiting for the store to calculate its address. The store cannot calculate its address until the load to R3 is complete. Since the load to R9 occupies load execution resources while stalling, the first load cannot complete. This causes a circular dependency which will indefinitely stall the processor. Luckily, we never encountered this case in the tests or benchmarks, however this should be corrected. The changes necessary to prevent this deadlock would not severely change the performance of this architecture.

Writeback hazards could be caused by loads and stores stalling for variables amounts of time. To avoid writeback hazards, this architecture allows for writeback from each execution pipe. This requires two additional writeback ports on the ROB.

G. *7-Stage OoO Processor with OoO Multi-Issue and Multiple In-Flight Memory Instructions*

Since multiple independent pipes with corresponding write-back ports are already in place, it is convenient to add multi-issue capabilities to the previous design. Multi-issue capabilities require additional ports to the scoreboard and more bypassing select signals from the scoreboard to correctly bypass values for issuing instructions. In addition, capabilities must be added to the IQ to select multiple instructions that use different functional units and multiple issue ports must be added to output the issued instructions. Multi-issue also prompts the creation of several bypassing multiplexers to select bypassing values.

4. Implementation

With high level functionality established, this section covers a number of details in implementing the architectures described above.

A. *7-Stage OoO Processor with OoO Single Issue and Single In-Flight Memory Instructions*

As mentioned before, two structures must be added to the 6-stage OoO processor: an IQ and a rename table. The IQ

Listing 1: Deadlock Example Code

```
lw  $r3 ,  0($r2)
sw  $r1 ,  0($r3)
lw  $r9 ,  0($r10)
```

TABLE I: Issue Queue Contents

Piped Data	**Valid**	**Speculative**	**Memory**	**Src0 Valid**	**Src0 Pending**	**Src0 Data**	**Src1 Valid**	**Src1 Pending**	**Src1 Data**
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

must track a set of necessary information to guarantee correct issuing of instructions, shown in Table I. Each entry must hold all information necessary for the instruction to execute; this includes the instruction register, PC+4 for branches, and the ROB write destination. This data is denoted as piped data in Table I.

The other bits required include a valid bit, a speculative bit, and a memory bit. The valid bit is set high on entry and low on exit and rollback. The speculative bit is set when an instruction is decoded after a branch in flight. We only use a single speculative bit for simplicity, so only one branch can be allowed in flight at a time. The system predicts all branches not taken, so when a branch is not taken, all speculative bits are cleared. When a branch is taken, all entries with the speculative bit set are invalidated. When a memory instruction enters the queue, a flag is set to stall any other memory instructions attempting to enter. When an instruction with a high memory bit leaves the issue queue, the flag is reset and the next memory instruction is allowed to enter.

Each entry in the issue queue stores information about each of its sources including a valid bit, a pending bit and a data field. The valid bit indicates if the instruction uses the source it is attached to. The pending bit indicates if that source is anywhere outside of the register file. The data field holds the 32 bit value of the data if the pending bit is not set. If the pending bit is set, then the data field contains the ROB entry which that data will be written to. The scoreboard provides a 16 bit (the size of the ROB) wide vector which uses one-hot encoding to indicate which ROB slots can be bypassed from the pipes or out of the ROB. Upon issue, if an instruction's source has a high pending bit, the scoreboard bypasses the value based on the ROB slot contained in the data slot. Priority is given to older instructions. As a reminder, instructions cannot issue if their sources cannot be bypassed or are not in the register file.

The issue queue keeps a head and tail pointer to keep track of the order that instructions were inserted. Instructions are inserted at the tail and issued from the head. This makes prioritizing older instructions simple, but brings up an issue of updating the head pointer, as there can be holes of invalid instructions between the head and tail. In order to solve this, we use a set of if-else statements in Verilog to check, from the head, the first instruction that is ready to issue.

The scoreboard, shown in Table II, contains shift registers which track the latency of each ROB slot. This tracks how soon the data will be available and shifts every cycle if the pipe is not stalling. The scoreboard also uses the combination of all in flight instructions' latencies to schedule around writeback hazards. Pending bits determines if the value for that ROB slot was written to the register file. The pending bits are cleared on commit of ROB slots. The scoreboard tracks if the next instruction to write the ROB entry is contained in the IQ, with the "In IQ" bit, to determine if values are available.

The register renaming table is located inside the scoreboard. The renaming table holds two mappings: register to ROB slot and ROB slot back to register. Keeping track of both of these mappings prevents the use of a content addressed lookup in one.

The register to ROB mapping table, shown in Table III, is indexed by register numbers and contains a field for the ROB slot. Each slot has a valid and speculative bit attached to it. The valid bit is set on ROB allocation and is reset on rollback and commit. The speculative bits have the same function to those contained in the IQ. The ROB slot to register mapping table, shown in Table IV, holds valid and speculative bits with the same function as those held in the register to ROB mapping.

The renaming tables both have previous entries which are filled with old values when a speculative mapping is written. This allows for successful rollback. In this architecture, we do not allow any more than one speculative instruction to write to the same register, the second writing instruction must stall.

ROB entries must also be adjusted to make room for pending and speculative bits. Speculative bits serve the same rollback purpose as in the IQ and scoreboard. Pending bits determine if the ROB slot is filled and can be committed. It should also be mentioned that the head pointer for both the ROB and IQ must be capable of moving more than one slot. We unfortunately did not have enough time to implement this in an elegant manner, and use a series of if/else statements to determine where the head should be updated to.

B. 7-Stage OoO Processor With OoO Single Issue and Multiple In-Flight Memory Instructions

To allow multiple in flight memory instructions, an LSQ must be implemented. The LSQ, shown in Table V holds enough information about loads and stores to determine if a RAW, WAR, or WAW hazard exists. This includes a valid bit, a speculative bit, a load/store bit, and a pending bit for each instruction. The valid bit is set on entry and set low on exit or rollback. The speculative bit plays the same role as in the IQ, scoreboard and ROB. The load/store bit declares if the instruction is a load or a store. The pending bit determines if the address has been calculated yet. If the address has been calculated, then an address field is filled.

TABLE II: Scoreboard ROB Slot Latency Tracking Table

Index	Pending	Latency	Functional Unit	In IQ
0				
1				
. . .	. . .	. . .	. . .	. . .
ROB Size - 1				

TABLE III: Scoreboard Register to ROB Slot Rename Table

Index	Valid	Speculative	ROB Slot	Previous Valid	Previous ROB Slot
0					
1					
. . .	. . .	. . .	. . .	. . .	. . .
31					

TABLE IV: Scoreboard ROB Slot to Register Rename Table

Index	Valid	Speculative	Register	Previous Valid	Previous Register
0					
1					
. . .	. . .	. . .	. . .	. . .	. . .
ROB Size - 1					

TABLE V: LSQ Table

Valid	Speculative	Load/Store	Pending	Length	Address	Write Data
. . .	. . .	. . .	. . .	. . .	. . .	. . .

Store instructions are provided with a data field to be filled so that loads can bypass from them. Each memory operation's length is also stored. The length is crucial since data cannot be bypassed from a matching instruction unless the lengths match. Finally, it important to note that the head must be capable of adjusting in a similar manner to that of the IQ's and ROB's heads. This was implemented in the same way. The IQ, ROB and LSQ all use a val/rdy interface to indicate when they should accept values.

C. 7-Stage OoO Processor with OoO Multi-Issue and Multiple In-Flight Memory Instructions

In a multi-issue setup, the IQ must be able to determine which pipes can be issued down. Each instruction in the IQ declares which pipe it will use after issue. The IQ selects one for each pipe (if available) assigning priority to older instructions. If a pipe is stalled, an instruction is not selected to issue down that pipe.

5. Experimental Methodology

Each of the processors described previously have been implemented in the Verilog hardware description language. The processors implement the PARCv2 [18] instruction set architecture. Icarus Verilog [19] was used to compile and simulate the processors.

Each processor was tested using a set of assembly test which verified the correct operation of each instruction. Additionally, assembly tests were written to exercise specific functionality or corner cases for the processors. The processors were also verified to provide the correct output on four benchmarks used in the evaluation. During the evaluation, the IQ size was set to 32 and the ROB size was set to 64.

Each of the benchmarks used in the evaluation is described in Table VI. The vvadd benchmark performs an addition of two vectors of integers and has a large amount of independent work, thus very few data hazards, and long strings of loads and stores to load values from the source vectors and store values to the destination vectors. The complx-mult benchmark performs multiplication on two sets of complex numbers which leads to a fair amount of independent work and large amounts of long latency multiply operations. The masked-filter benchmark applies a masked filter to a vector of integers resulting in many long latency multiply and divide operations in addition to a fair amount of dependent work. Lastly, bin-search performs a binary search of a set of integer keys on an ordered list of integers and returns the index of the key in the

TABLE VI: Benchmark Descriptions

Benchmark	Description	Characteristics
vvadd	Performs addition on two vectors of integers	Very few data hazards. Lots of independent additions. Strings of loads and stores.
cmplx-mult	Performs multiplication on two sets of complex numbers	Many long latency multiplication instructions. Fair amount of independent work.
masked-filter	Applies a masked filter to a vector of integers	Good amount of dependent operations. Many long latency multiplication and division instructions.
bin-search	Performs a binary search of a set of integer keys on an ordered list of integers and returns the index	Many dependent operations

list. This results in many dependent operations, almost every operation depends on the next.

6. Results

The results from simulating the processors implemented in this lab with the four benchmarks described are presented in Figure 4. Comparing performance across each of these benchmarks reveals several trends. In general, processors which need to stall less achieve a higher IPC than those which stall more often. The two-piped superscalar processor possess a large advantage due to the fact that it is able to fetch multiple instructions from memory. This section describes in detail the trade-offs which are reflected in the performance of each architecture discussed in this paper.

The 5-stage stalling processor (pv2stall) comes in last on performance on most benchmarks. Since this processor stalls on all dependencies, it performs poorly an all benchmarks except for vvadd. The vvadd benchmark has very few dependencies, so the 5-stage stalling processor maintains good performance in this case. We would expect the 5-stage bypassing processor (pv2byp) to outperform pv2stall on every benchmark, and it does. Since the bypassing processor allows for dependencies to be bypassed with no other microarchitecural changes, performance can only improve.

The superscalar processor outperforms all other processors since it is capable of fetching two instructions at the same time. We had planned to improve the 7-stage OoO processor with OoO multi-issue and multiple in-flight memory instructions (pvmiooo) to support dual-fetch, however, we were unable to develop this capability to pass the benchmarks under the time constraint. We would expect a large improvement if we completed this, as much of the performance is limited by the ability to fill the IQ.

The 6-stage OoO processor with in-order single issue (pv2ooo) underperformed compared to the bypassing processor on all counts because it must stall on writeback hazards, its performance is comparable, otherwise.

The 7-stage OoO processor with OoO single issue and single in-flight memory instruction (pv2eooo) significantly under-performs most of the microarchitectures we have discussed. Since the processor only allows a single memory instruction in flight, it performs quite poorly on sequential memory operations of which vvadd, complex multiply and masked-filter possess many. Additionally, pv2eooo does not allow for multiple branches or speculative store instructions causes the processor to lose some performance compared to the in-order issue processors.

the 7-stage OoO processor with OoO single issue and multiple in-flight memory instructions (pv2mpooo) improves on the pv2eooo's performance on all counts. Since the processor can issue multiple memory instructions at once, many stalls are avoided. The pv2mpooo is also capable of avoiding structural hazards in the writeback stage since it can write back three instructions at the same time. This significantly speeds up the processor on benchmarks with a variety of operations traveling down different pipes (such as vvadd, complex multiply and masked-filter).

The pv2miooo consistently improves upon the mp2mpooo's performance since more instructions can be issued. The processor can still only allow a single branch in flight, but otherwise has comparable and even better performance than pv2byp. However, a performance bottle neck now lies in the fetch stage, since the processor is capable of issuing most instructions as soon as their operands are available, the IQ is usually close to empty. The addition of dual fetch will allow this processor to achieve very high performance.

We also explored varying the size of the IQ and ROB (ROB just sized to ensure it can hold the number of instructions possible in the IQ plus the number of instructions possible in-flight), however, this turned out to not be very interesting. No performance gain was achieved by increasing the size of the IQ past four, thus our bottleneck was in the ability to fill the IQ fast enough. If we had been able to get pv2miooo working with dual-fetch, this study may have been a bit more interesting as filling the IQ may not have been the bottleneck. The bottleneck may have lied in the ability to find independent work.

7. Conclusion

In our study of each microarchitecture described in this report, it is clear that a variety of approaches can help improve ILP utilization. Clearly, avoiding cases where instructions must stall, and providing structures to stop these stalls improves IPC dramatically. The pv2stall processor showed extremely poor performance compared to processors which conquered data dependencies.

We can also draw the conclusion that once data dependencies are removed from an in order issue microarchitecture, out of order issue architecture can improve performance by effectively reducing the number of stalls, allowing younger instructions to fill stall slots. However, it is important to note

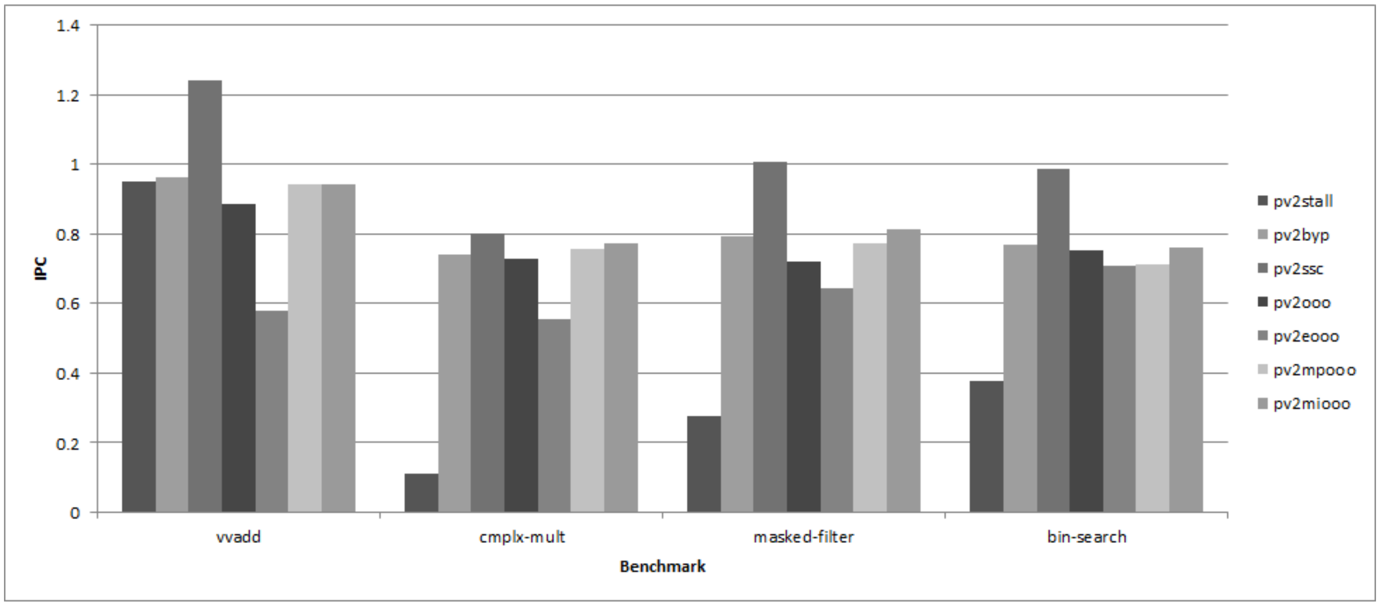

Fig. 4: IPC for different processors on each benchmark

that the number of stalls added as a result of out of order processing should not outweigh the number of stalls prevented as is the case in the pv2eooo processor. This increased number of stalls lead to a degradation in performance. Upon observation of the pv2mpooo processor's performance, We can see a dramatic increase in performance when most stalling cases are avoided.

Finally, to increase the throughput of the processor, multi-issue is the next step in improving performance. However, a multi-issue structure provides little performance gain if the bottle neck of single-fetch is not opened up. The pv2miooo processor does improve performance with multi-issue over the pv2mpooo, but reflects the limited power of multi-issue without multi-fetch.

In general, each of these processors represents an evolution in processor design to take advantage of ILP. It is clear from gains in performance why modern processors utilize structures to increase ILP utilization.

References

[1] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, 1974.

[2] G. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, p. 4, 1965.

[3] R. Rojas, "Konrad zuse's legacy: The architecture of the z1 and z3," *IEEE Ann. Hist. Comput.*, vol. 19, no. 2, pp. 5–16, Apr. 1997.

[4] H. Brearley, "Illiac ii-a short description and annotated bibliography," *Electronic Computers, IEEE Transactions on*, vol. EC-14, no. 3, pp. 399–403, 1965.

[5] E. Bloch, "The engineering design of the stretch computer," in *IRE/AIEE/ACM Eastern Joint Computer Conference*, December 1959, pp. 48–58.

[6] J. E. Thornton, "Parallel operation in the control data 6600," in *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, ser. AFIPS '64 (Fall, part II), 1965, pp. 33–40.

[7] S. McGeady, "Inside intel's i960ca superscalar processor," *Microprocess. Microsyst.*, vol. 14, no. 6, pp. 385–396, Jul. 1990.

[8] J. Smith and G. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, 1995.

[9] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[10] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The ibm system/360 model 91: machine philosophy and instruction-handling," *IBM J. Res. Dev.*, vol. 11, no. 1, pp. 8–24, Jan. 1967.

[11] R. Hintz and D. P. Tate, "Control data star-100 processor design," *Compcon*, 1972.

[12] R. M. Russell, "The cray-1 computer system," *Commun. ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.

[13] J. Larson, "Multitasking on the cray x-mp-2 multiprocessor," *Computer*, vol. 17, no. 7, pp. 62–69, 1984.

[14] H. B. Bakoglu, G. F. Grohoski, and R. Montoye, "The ibm risc system/6000 processor: Hardware overview," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 12–22, 1990.

[15] J. Smith, G. E. Dermer, B. Vanderwarn, S. D. Klinger, C. Rozewski, D. L. Fowler, K. R. Scidmore, and J. Laudon, "The astronautics zs-1 processor," in *Computer Design: VLSI in Computers and Processors, 1988. ICCD '88., Proceedings of the 1988 IEEE International Conference on*, 1988, pp. 307–310.

[16] J. A. Fisher, "Very long instruction word architectures and the eli-512," *SIGARCH Comput. Archit. News*, vol. 11, no. 3, pp. 140–150, jun 1983.

[17] H. Sharangpani and H. Arora, "Itanium processor microarchitecture," *Micro, IEEE*, vol. 20, no. 5, pp. 24–43, 2000.

[18] J. Kim, "Parc instruction set architecture," August 2011. [Online]. Available: http://www.csl.cornell.edu/courses/ece4750/handouts/ece4750-parc-isa.txt

[19] "Icarus verilog." [Online]. Available: http://iverilog.icarus.com/