

Simple clusters with in-order issue.

Anu Vaidyanathan

1. Abstract

Cluster based microarchitectures are designed to distribute the hardware resources between what are called processing elements. Instead of having a single cluster with all the resources in it, if the resources were to be distributed, we would have reduced complexity and comparable performance. The design of clustered microarchitectures involves using a small number of clusters with their own copies of functional units, register files and issue windows. The issue and execution logic is also replicated for each cluster. The renamed instructions are steered to one of the clusters by some steering algorithm. Communication of dependences between clusters is achieved by wires that run between clusters. These are the potential bottlenecks to the performance of this kind of architecture. The front-end of the clustered architecture is the same as the conventional architecture but the presence of the steering logic distinguishes it from the conventional architecture. Distribution of instructions between clusters however simplifies a lot of the complexity and aims at achieving a faster clock cycle for the processor on the whole. FIFO based microarchitecture is the nomenclature used to describe those cases where the issue-windows within the individual clusters is implemented as FIFO queues. Each cluster can have one or many FIFO queues from where instructions are issued. The aim of this study was to design and verify the performance of very simple clusters with in order issue. The 'simple clusters' that were designed basically had one FIFO in them and so the issue-width for each cluster was one. The instructions were steered to these clusters / FIFO's using a data-dependence based heuristic. There are three classes of instruction steering algorithms. There is the functional unit dependence based steering algorithm. Then there is the control-dependence based steering algorithm and then finally there is the data-dependence based steering algorithm. The data-dependence based steering algorithms have been shown to have the highest performance and were therefore used in this study. The IPC of the single FIFO architecture implemented in this study showed a degradation of a maximum of 32% in one of the benchmarks and the others were between 16%-25%, when compared a 4-way,out-of-order superscalar processor. Around 7%-12% of the total instructions were found to communicate register values with the other clusters.

2. Introduction

Cluster based microarchitectures are designed to distribute the hardware resources between what are called processing clusters. Instead of having a single cluster with all the resources in it, if the resources were to be distributed, we would have reduced complexity and comparable performance. The design of clustered microarchitectures involves using a couple of clusters with their own copies of functional units, register files and issue windows. The issue and execution logic is also replicated for each cluster. The renamed instructions are steered to one of the clusters by some steering algorithm. Communication of dependences between clusters is achieved by wires which run between clusters. These are the potential bottlenecks to the performance of this kind of architecture. The front-end of the clustered architecture is the same as the conventional architecture but the presence of the steering logic distinguishes it from the conventional architecture. Distribution of instructions between clusters however simplifies a lot of the complexity and aims at achieving a faster clock cycle for the processor on the whole.

FIFO based microarchitecture is the nomenclature used to describe those cases where the issue-windows within the individual clusters is implemented as FIFO queues. Each cluster can have one or many FIFO queues from where instructions are issued. Palacharla, Jouppi and Smith[PJS] proposed a dependence based microarchitecture. By using FIFO queues instead of the conventional issue-window, these architectures saw a reduction in the complexity associated with the wakeup and select logic of the processor. Wakeup and select logic have been identified(in the same study) as being the most critical structures in ooo superscalar processors. This study showed that the FIFO based architectures achieved a comparable IPC performance and much better clock rates than the conventional single-cluster design. Jerry Tseng [JERRY]. Studied the utilization of the FIFO queues inside the FIFO-based microarchitecture. Based on this analysis, he proposed an optimal instruction steering algorithm. His results show that both the optimized FIFO-based microarchitecture and the FIFO-based clustered microarchitecture with his steering algorithm can achieve better IPC performance than the original implementations of [PJS][Pal98].

The report is organized in the following manner. Section 2 summarizes previous work in this field. Section 3 presents the results of this study along with the architecture simulated. Section 4 compares these results with a conventional 4-way ooo machine. Section 5 discusses the future work that could be done using this model as a baseline and results from some studies which could be relevant to this work. Section 6 draws conclusions and wishes Wisconsin good bye.

2.1 Historical Perspective

Palacharla, jouppi and smith [ref] studied FIFO based microarchitectures which used a dependence based steering algorithm. Inter-cluster communications are the most significant performance overhead in clustered architectures. Wire delays are going to be very critical in future architectures and therefore a designer would typically work on decreasing the communication or the traffic between clusters. This can be achieved if the instruction steering algorithms are designed such that most or all of the dependent instructions go to the same cluster. There have been several steering algorithms proposed in the study by [PJS]. These can be enumerated as follows:

Single Window, Multiple Execution Clusters, Execution-driven steering.

Multiple windows, Dispatch-driven steering:

FIFO steering.

Round Robin steering.

Random Steering.

This was followed by Jerry Tseng's work, which looked at a different steering algorithm for FIFO-based issue-windows. This study was conducted to look at simple clusters with in-order issue. This is primarily aiming at increasing the clock speeds by reducing the hardware complexity associated with primary clusters. Steering algorithms primarily look at optimizing the following parameters:

- Inter-cluster communications.
- Number of register values that have to be communicated every cycle.

3. Heuristic and Results

3.1 Design:

The algorithm used to steer instructions in this study has the following steps:

If an instruction has the form $rd \leftarrow rs,rt,rj$.

- Identify the creators of rs,rt and rj.
- If there are no creators, steer the instruction to the shortest FIFO.
- If there is a creator for one of the three registers:
 - Check to see if the creating instruction has already issued.
 - If it has, check to see if the FIFO it issued from is full.
 - If that FIFO is not full, steer this instruction to that FIFO.
 - If that FIFO is full, steer this instruction to the shortest FIFO.
 - If the creating instruction has not issued, check to see if the FIFO it was steered to is full.
 - If that FIFO is not full, steer this instruction to that FIFO.
 - If that FIFO is full, find the relative position of the creating instruction within the FIFO and steer this instruction to the FIFO which is as long as the position in which the creator is sitting.
 - If the creator is in the last slot of a FIFO, steer the instruction to the Longest FIFO available.
- If there is a creator for more than one of the three registers:
 - Apply the steps applied to the case when we had one creator.
 - If the creating instruction(s) have issued, try to steer it to the FIFO from which the creator(s) issued.
 - If the FIFO of the creator(s) are full, steer the instruction to the shortest FIFO.
 - If the creator's have not issued, find the relative positions of the creators within their respective FIFO's.
 - Depending on which creator is at a latter position, try to steer the instruction to that FIFO or to a FIFO which is as long as the position in which the creator at the latter relative position is sitting.

3.2 Notes on the algorithm:

There are four FIFO's, each 16-slots deep. When starting up or deciding which FIFO's to start filling, we go in-order (i.e, FIFO[0] first, FIFO[1] next and so on.) This scheme can be seen to an extent in some of the results that have been graphed. For example, critical delays and FIFO empty numbers taper and increase from FIFO[0] to FIFO[3] respectively. This is because FIFO[0] gets filled first and more often has a greater number of instructions in it. FIFO empty increases because every time the issue window starts filling up, instructions first go to FIFO[0] and then fill up in the other FIFO's/clusters in order. Anytime an instruction has two creators, both of which have issued, the algorithm tries to steer it to the FIFO of the second creator first. The algorithm takes into account the relative position of the creators of an instructions dependences and tries to steer it behind he creator which is further behind in the FIFO it is sitting in. This aims at reducing the number of times an instruction proceeds to the head of the FIFO, does not have its dependences satisfied and consequently holds up the instructions behind it. If an instruction's creators (i.e creators of input dependences), have already issued, the algorithm first tries to steer the dependent instruction to the same FIFO as the creator issued from. By this, we are effectively cutting down communications between the FIFO's/clusters and can also reduce traffic between copies of the register file*. Simplescalar has a particular way of dealing with loads and stores. There is a separate queue called the Load/Store queue out of which the actual memory portion of the Loads and Stores issue. The effective address computation is done separately. When checking to see if a load instruction can issue, the entire LSQ is scanned for stores which might have address or data dependences. If a load is going to be loading a value from and address a store is going to be writing to, that load instruction cannot issue. Furthermore, if the store instruction that is creating this sort of dependency is in another cluster, we have to take this into account as a communication. Since the design seeks to model each cluster as having its own Load/Store queue, these are in a way communications between clusters and these were found to contribute to 3%-4% of communications. Each cluster was modeled to have its own set of functional units (by having an array of functional unit sets). All these sets were identical. Simplescalar also divides up Load/Store instructions into two parts: Effective Address computation and the memory operation. While previous implementations

[JERRY] had the split-up instruction occupy two slots in the FIFO's, here they occupy only one (the EA computation instruction). There were a number of modules of code to execute collection of statistics efficiently.

3.3 Verification: Verification was done using the following general methods:

- Assertion of invariants. (eg. An instruction that is not at the head of the FIFO cannot issue, FIFO lengths cannot be negative and the memory operation parts of Loads/Stores cannot occupy a slot in the FIFO).

Printouts and comparison of commit streams of this implementation vs. the simplescalar simulation of a 4-way ooo machine.

Commit Stream from simplescalar:

PC	seq #	instruction
4194624	1	lw r16,0(r29)
4194632	3	lui r28,0x1002
4194640	4	addiu r28,r28,-8464
4194648	5	addiu r17,r29,4
4194656	6	addiu r3,r17,4
4194664	7	sll r2,r16,2
4194672	8	addu r3,r3,r2
4194680	9	addu r18,r0,r3
4194688	10	sw r18,-31580(r28)
4194696	12	addiu r29,r29,-24

Commit Stream from this simulator:

PC	FIFO #	seq #	instruction
4194624	0	1	lw r16,0(r29)
4194632	1	3	lui r28,0x1002
4194640	1	4	addiu r28,r28,-8464
4194648	2	5	addiu r17,r29,4
4194656	0	6	addiu r3,r17,4
4194664	0	7	sll r2,r16,2
4194672	0	8	addu r3,r3,r2
4194680	0	9	addu r18,r0,r3
4194688	0	10	sw r18,-31580(r28)
4194696	1	12	addiu r29,r29,-24

Printouts and comparison of issue streams of this implementation vs. the simplescalar simulation of a 4-way ooo machine.

This involved checking to see if the same instructions were mispredicted and squashed using the same branch prediction parameters/methodology. There was also a check to see that no two instructions issued from any given FIFO at the same time since we are simulating in-order issue within a particular FIFO.

Issue Stream from this simulator(Each issue cycle is demarcated by a solid black line):
 Note: Replication of FIFO# implies loads/stores executing the memory operation.

PC	FIFO#	seq#	Instruction	
4621344	1	1772061	sll	r3,r2,2
4621352	1	1772062	addu	r3,r3,r5
4621592	0	1772072	addiu	r16,r16,1
4621360	1	1772063	lw	r3,0(r3)
4621600	0	1772073	bgtz	r16,0x468418
4621336	2	1772074	lw	r2,40(r29)
4621360	1	1772064	lw	r3,0(r3)
4621368	1	1772065	addiu	r4,r2,1
4621384	0	1772082	sw	r4,40(r29)
4621336	2	1772075	lw	r2,40(r29)
4621376	1	1772066	andi	r2,r3,63
4621384	0	1772068	sw	r4,40(r29)
4621392	1	1772069	bne	r2,r9,0x468468
4621344	2	1772076	sll	r3,r2,2
4621592	0	1772087	addiu	r16,r16,1
4621352	2	1772077	addu	r3,r3,r5
4621600	0	1772088	bgtz	r16,0x468418
4621416	1	1772070	andi	r2,r3,192
4621336	3	1772089	lw	r2,40(r29)
4621424	1	1772071	beq	r2,r8,0x468518
4621360	2	1772078	lw	r3,0(r3)
4621384	0	1772097	sw	r4,40(r29)
<<<<<<< MISPREDICAT >>>>>>>>>				
4621424	1	1772071	beq	r2,r8,0x468518
***** SQUASH *****				
4621424	3	1772101	beq	r2,r8,0x468518
***** SQUASH *****				
4621416	3	1772100	andi	r2,r3,192
***** SQUASH *****				
4621392	3	1772099	bne	r2,r9,0x468468
***** SQUASH *****				
4621384	0	1772098	sw	r4,40(r29)
***** SQUASH *****				

There was one unexpected result that was the number of instructions that were totally independent of each other and waiting to issue in the ready queue and could not do so because they were not at the head of the FIFO. Some snapshots showed that there were as many as 34-38 instructions at any give time that could potentially issue but were not because they were not at the head of the queue. This was verified for correctness by maintaining a history of what actual instructions were scanned for issue before re-enqueing them because they were not at the head. Once the count of these instructions reached a number like 30 (which was the worst case for GO for a 5X5 matrix), these instructions were dumped in a file and checked for correctness. This has been included below.

The fields are from the register update unit station or a node of the register update unit.

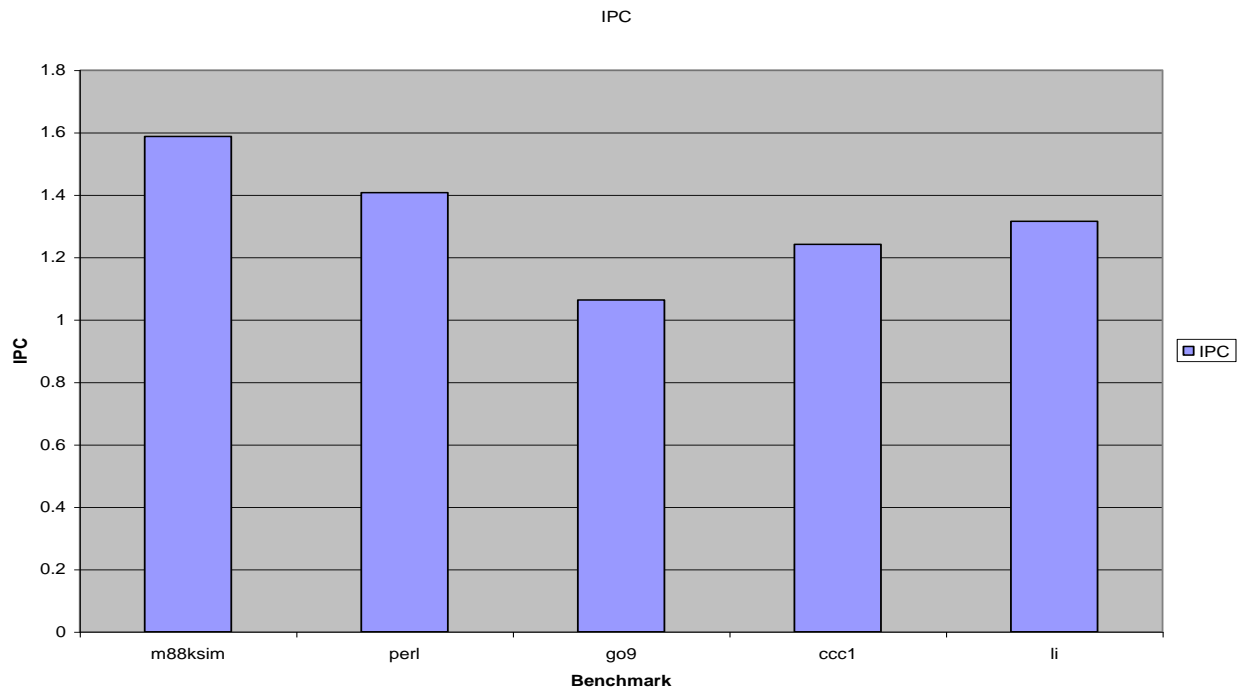
PC	FIFO #	seq #	in_LSQ	in_fifo	issued	queued	instruction
4711168	1	334547	0	1	0	1	jal 0x47e5a0
4714408	2	334533	0	1	0	1	jr r31
4714024	1	334470	0	1	0	1	lw r3,0(r13)
4714032	1	334472	0	1	0	1	addiu r16,r16,1
4714088	0	334481	0	1	0	1	lui r3,0x1008
4714136	3	334488	0	1	0	1	addiu r13,r13,4
4714144	0	334489	0	1	0	1	addiu r5,r5,4
4714152	0	334490	0	1	0	1	addiu r8,r8,-1
4714160	2	334491	0	1	0	1	addu r25,r25,r14
4714168	3	334492	0	1	0	1	addu r24,r24,r14
4714176	2	334493	0	1	0	1	addiu r15,r15,4
4714184	2	334494	0	1	0	1	addiu r9,r9,1
4714208	0	334497	0	1	0	1	addiu r30,r30,20
4714216	3	334498	0	1	0	1	addiu r22,r22,20
4714224	0	334499	0	1	0	1	addiu r11,r11,1
4714248	3	334502	0	1	0	1	sll r3,r19,1
4714272	1	334505	0	1	0	1	lui r2,0x1008
4714304	1	334510	0	1	0	1	lui r1,0x1008
4714328	1	334514	0	1	0	1	lw r30,88(r29)
4714336	0	334516	0	1	0	1	lw r23,84(r29)
4714344	1	334518	0	1	0	1	lw r22,80(r29)
4714352	2	334520	0	1	0	1	lw r21,76(r29)
4714360	0	334522	0	1	0	1	lw r20,72(r29)
4714368	1	334524	0	1	0	1	lw r19,68(r29)
4714376	2	334526	0	1	0	1	lw r18,64(r29)
4714384	0	334528	0	1	0	1	lw r17,60(r29)
4714392	0	334530	0	1	0	1	lw r16,56(r29)
4714400	1	334532	0	1	0	1	addiu r29,r29,96
4711088	2	334534	0	1	0	1	lui r2,0x1001
4711120	1	334539	0	1	0	1	lui r2,0x1001

*This has been elaborated on in the Future Work section.

4. Results

The results of this study have been tabulated below:

Benchmark	Instructions Committed	Simulation Cycles	IPC	Communication
m88ksim	108850300	68566508	1.5875	14248985
perl	41427098	29430648	1.4076	3509385
go9	132970003	125017404	1.0636	17392752
ccc1	264897865	213341448	1.2417	28803637
li	956867545	727364909	1.3155	73130583



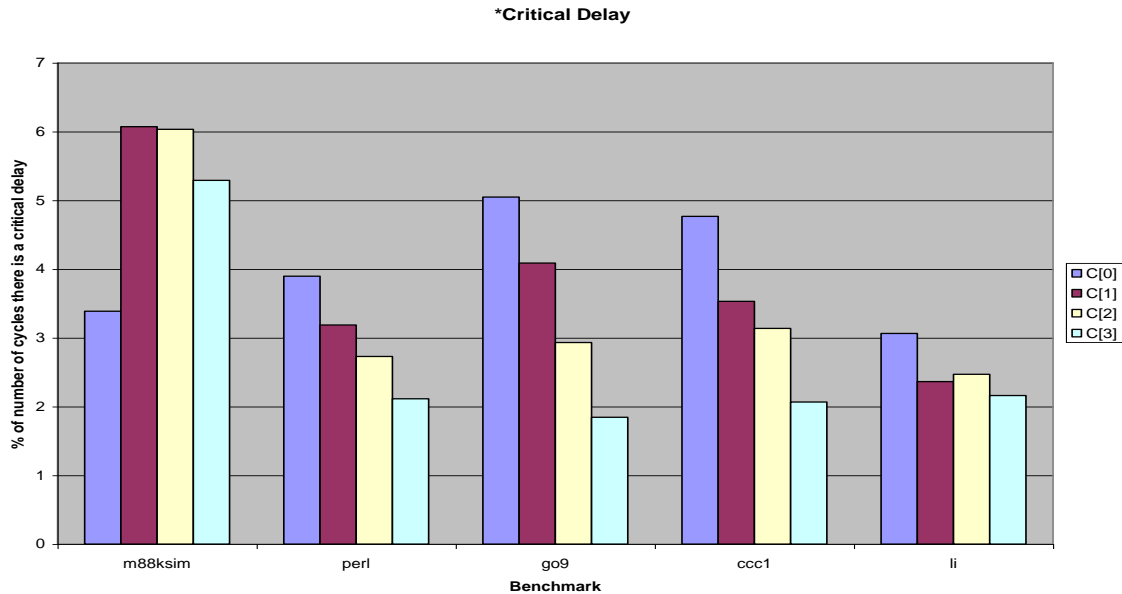
4.1 Critical Delays:

These are defined as the delays due to an instruction at the head of a FIFO which is waiting to issue but is waiting for a register value to be communicated from another cluster. This potentially is holding up the rest of the instructions behind it to an extent.

Benchmark	C[0]	C[1]	C[2]	C[3]
m88ksim	2322612	4162195	4137397	3626781
perl	1146598	938006	803075	621706
go9	6310692	5109421	3666445	2306194
ccc1	10168389	7533991	6692777	4408480
li	22277887	17185992	17960781	15705923

Percentage of cycles in the simulation that see critical delays: This is calculated by dividing the critical instruction numbers for each FIFO by the total number of cycles.

Benchmark	C[0]	C[1]	C[2]	C[3]
m88ksim	3.387385573	6.070303303	6.034136958	5.289435
perl	3.895931887	3.187174132	2.728703085	2.112444
go9	5.047850778	4.086967763	2.932747668	1.844698
ccc1	4.766251047	3.531423955	3.137119891	2.066396
li	3.06282125	2.362774419	2.469294405	2.159291

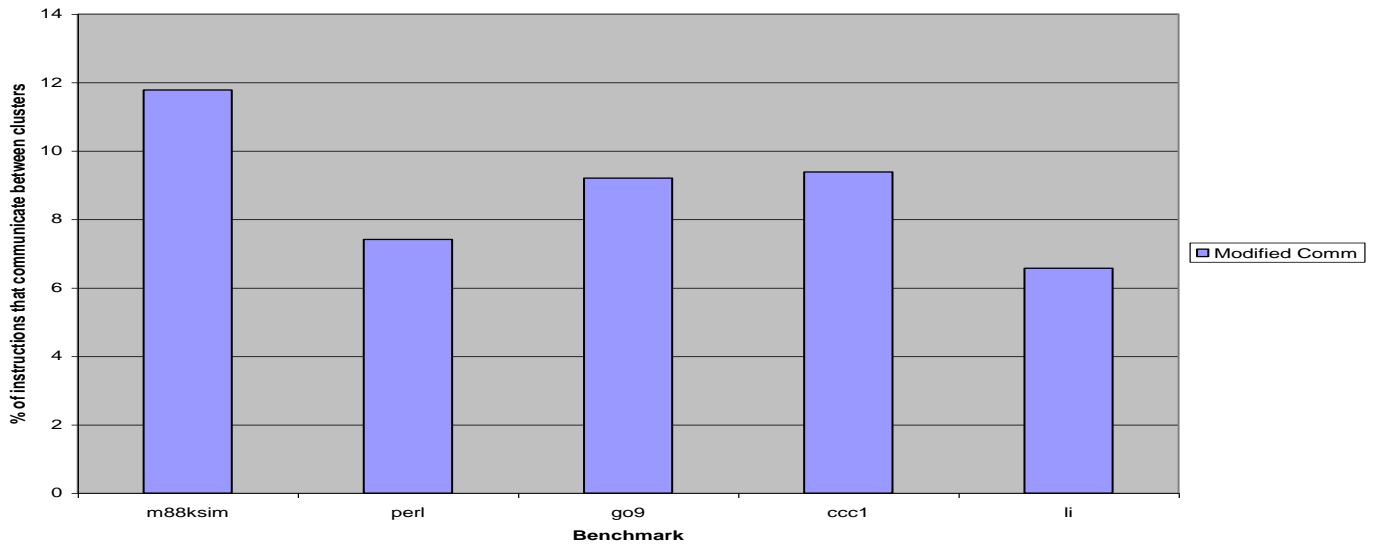


4.2 Inter-Cluster Communication: This is the communication of register values between the single FIFO clusters. Here we have five sets of data:

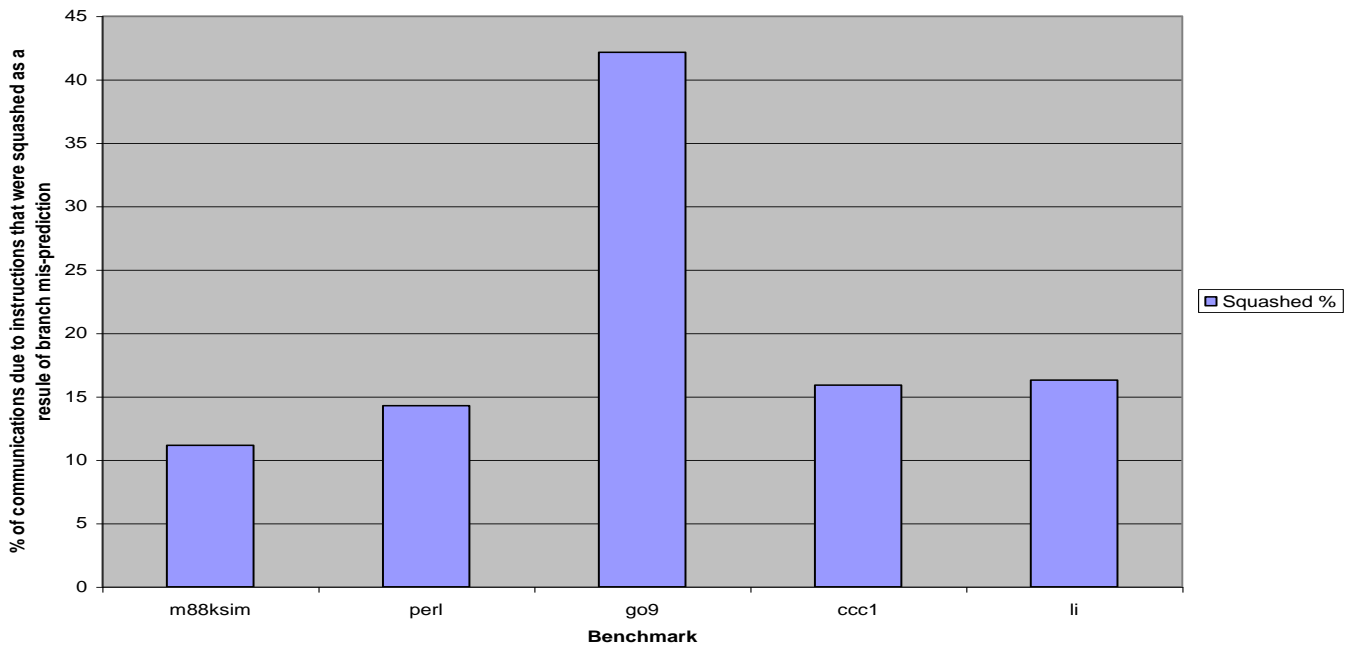
- Number of communications (raw numbers).
- Number of communications that occur but the instructions that communicate never retire because they are squashed as a result of a mis-predicted branch.
- Actual Communication (which is a difference of total and squashed).
- Percentage of instructions that communicate between clusters.
- Percentage of the actual communications that were caused by instructions that were squashed.

Benchmark	Communication	Squashed Insn.	ActualComm	Actual Comm %	Squashed %
m88ksim	14248985	1430416	12818569	11.77632859	11.1589367
Perl	3509385	438686	3070699	7.412295691	14.28619347
Go9	17392752	5157450	12235302	9.201550518	42.15220842
Ccc1	28803637	3951096	24852541	9.381933297	15.89815705
Li	73130583	10246508	62884075	6.571868314	16.29428118

Inter-Cluster Communication



Percentage of bad communications

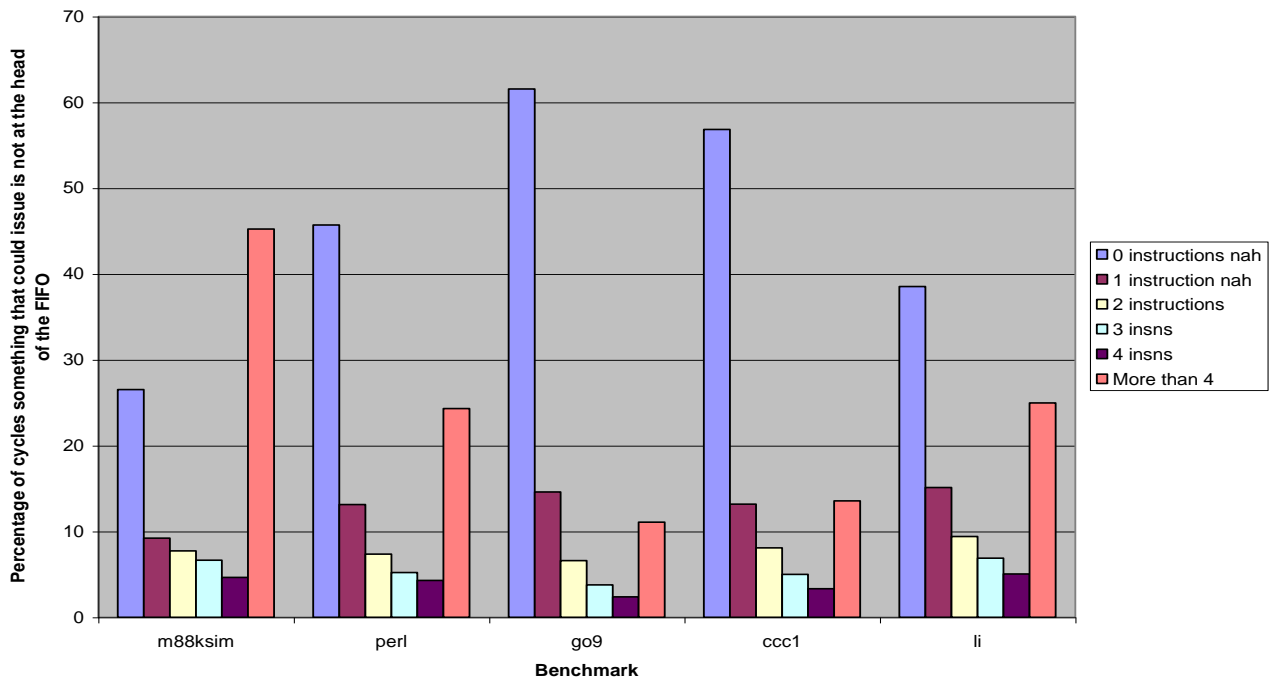


4.3 Not at Head Of FIFO Histogram:

This is a histogram of the number of instructions in the ready queue that were scanned before an instruction, which was at the head of the FIFO was found to issue. It is seen that there are cases when 34-38 instructions are totally independent of each other and waiting in the ready queue for issue. They are unable to issue as they are not at the head of the FIFO and we are essentially implementing an in-order paradigm.

Benchmark	0 instructions	1 insns	2 insns	3 insns	4 insns	More than 4
M88ksim	18192565	6318439	5296928	4558374	3187078	31013125
perl	13453596	3863017	2164327	1536089	1260342	7153278
go9	76950459	18253685	8268247	4712884	2978670	13853460
ccc1	121282102	28104296	17278895	10664171	7092826	28919159
li	280274308	110030033	68501001	50169589	36697253	181692726

Histogram of instructions in the readyq not at the head of the FIFO



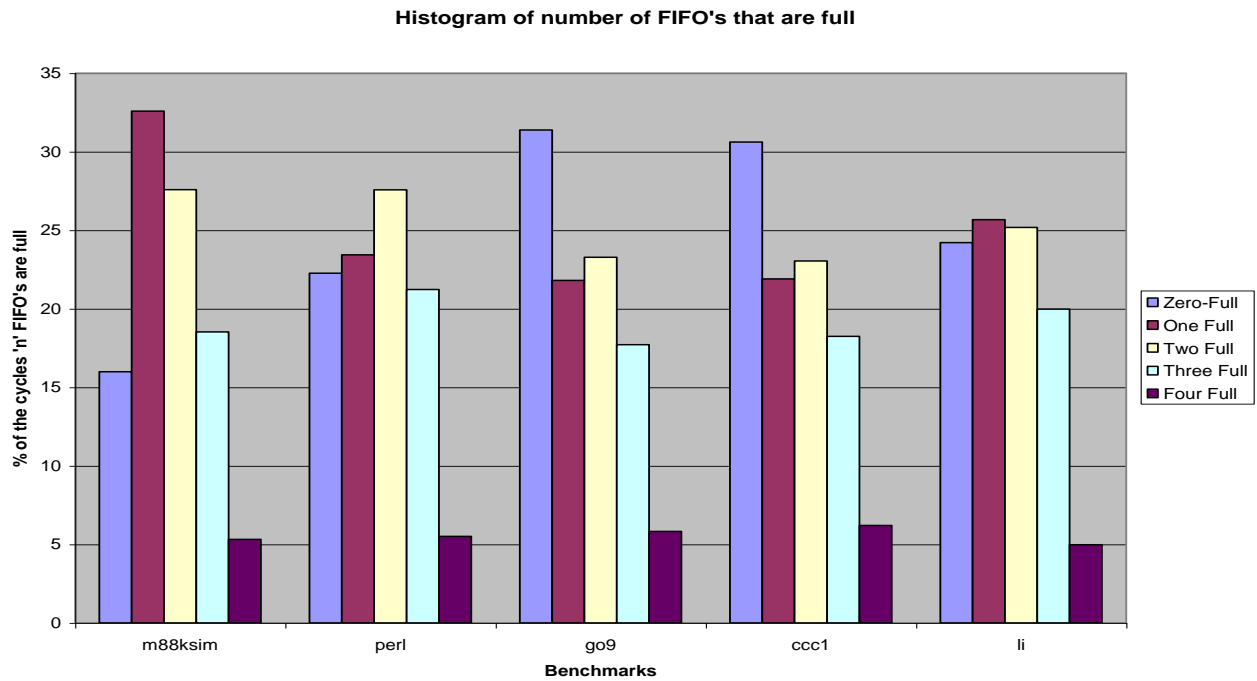
4.4 Histogram of the number of FIFO's that are full over all cycles of simulation: At any given time, either none, one, two, three or all the four FIFO's can be full. Here we have two sets of data:

- The number of FIFO's that are full at any given time (raw numbers).
- The percentage of the total number of cycles over which these number of FIFO's are full.

Benchmark	Zero-Full	One Full	Two Full	Three Full	Four Full
M88ksim	10960567	22338959	18916330	12703227	3647426
perl	6554066	6896197	8113811	6246511	1620064
go9	39232331	27256996	29106551	22138098	7283429
ccc1	65320440	46712316	49154121	38923283	13231289
li	176068977	186740825	183097904	145288857	36168347

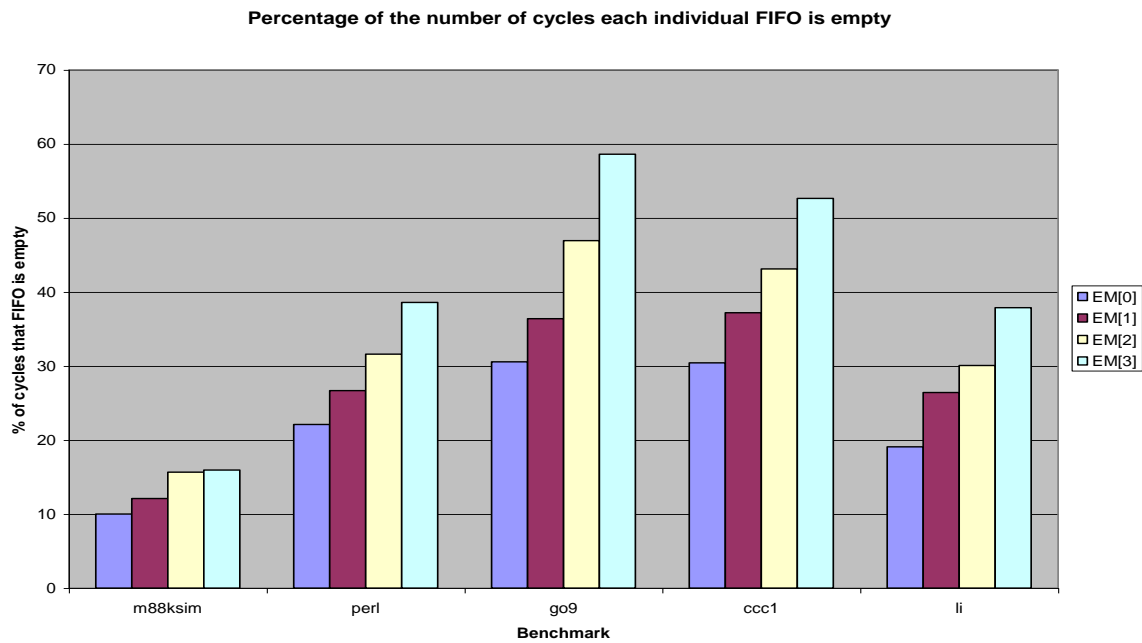
Percentage of Cycles which the FIFO's are cumulatively FULL

Benchmark	Zero-Full	One Full	Two Full	Three Full	Four Full
m88ksim	15.98531	32.57999	27.58829	18.52687	5.319545
perl	22.26953	23.43203	27.56926	21.22451	5.504683
go9	31.3815	21.80256	23.282	17.70801	5.825932
ccc1	30.61779	21.89557	23.04012	18.24459	6.201931
li	24.20642	25.67361	25.17277	19.97469	4.972517



4.5 FIFO Empty: This is a tabulation of the percentage of total simulation cycles that each individual FIFO is empty. Each FIFO is 16-deep and these could be snapshots of the issue stream immediately after a branch mis-predict when all the instructions have been flushed.

Benchmark	EM[0]	EM[1]	EM[2]	EM[3]
m88ksim	10.00177375	12.09954137	15.66970714	15.9356
perl	22.11080096	26.67267129	31.59801306	38.57597
go9	30.55596243	36.39997516	46.94085393	58.60467
ccc1	30.41370048	37.18436091	43.11622419	52.65082
li	19.08536682	26.4202439	30.08096257	37.88488



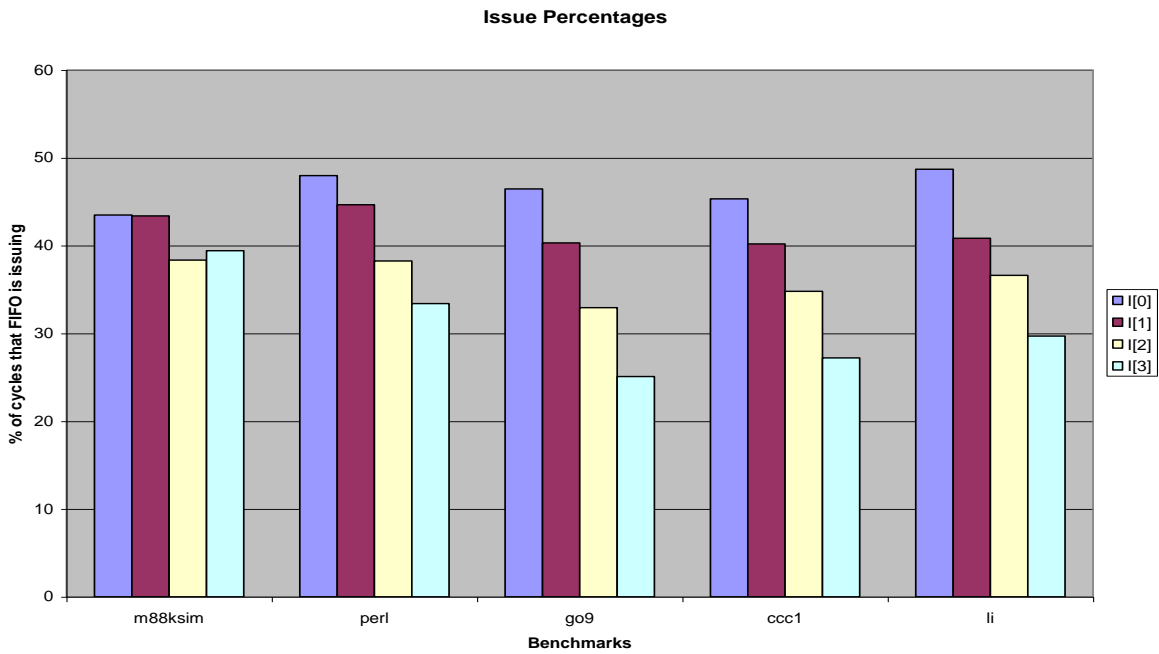
4.6 Issue Percentages: Here we have two sets of data:

- The number of cycles something issues out of a particular FIFO (raw numbers).
- The percentage of the total simulation cycles something issues out of a particular FIFO.

Benchmark	I[0]	I[1]	I[2]	I[3]
m88ksim	29810492	29746818	26287712	27025982
perl	14119040	13141602	11257196	9825770
go9	58084162	50401998	41175017	31356931
ccc1	96704191	85739075	74208491	58063806
li	354179385	297066501	266235386	215995320

Percentage of the total cycles something issued out of that particular FIFO:

Benchmark	I[0]	I[1]	I[2]	I[3]
m88ksim	43.47675399	43.38388941	38.33899781	39.41572
perl	47.9739352	44.65277829	38.24990873	33.38618
go9	46.46086076	40.31598512	32.93542793	25.08205
ccc1	45.32836535	40.18866273	34.78390706	27.21637
li	48.69349354	40.84146724	36.60272618	29.69559



For each of the following value, there are two sets of data being presented:

- Raw numbers
- Percentage of the total cycles that these numbers hold for.

4.7 FIFO FULL: The number of times each individual FIFO was full (that is, all 16 entries were consumed).

NOT ISSUE: The number of times nothing issued out of a particular FIFO.

FIFO Full:

Benchmark	Full[0]	Full[1]	Full[2]	Full[3]
m88ksim	15402808	7350451	13820119	8451634
Perl	1178002	1027921	876340	1229772
go9	3766894	4615504	4519360	3208076
ccc1	8634013	8662828	9250864	7364819
Li	48011575	47417153	66226844	36639679

Percentage of the total cycles that particular FIFO is full.

Benchmark	FULL[0]	FULL[1]	FULL[2]	FULL[3]
m88ksim	22.46404032	10.72017697	20.1557866	12.32618409
Perl	4.002636979	3.492688982	2.97764426	4.178542042
go9	3.013095681	3.691889171	3.61498468	2.566103516
cccl	4.047039654	4.060546172	4.33617756	3.452127596
Li	6.600754918	6.519032251	9.10503699	5.037317383

FIFO does not issue:

Benchmark	NI[0]	NI[1]	NI[2]	NI[3]
m88ksim	38756017	38819691	42278797	41540527
Perl	15311609	16289047	18173453	19604879
go9	66933243	74615407	83842388	93660474
cccl	116637258	127602374	139132958	155277643
Li	373185525	430298409	461129524	511369590

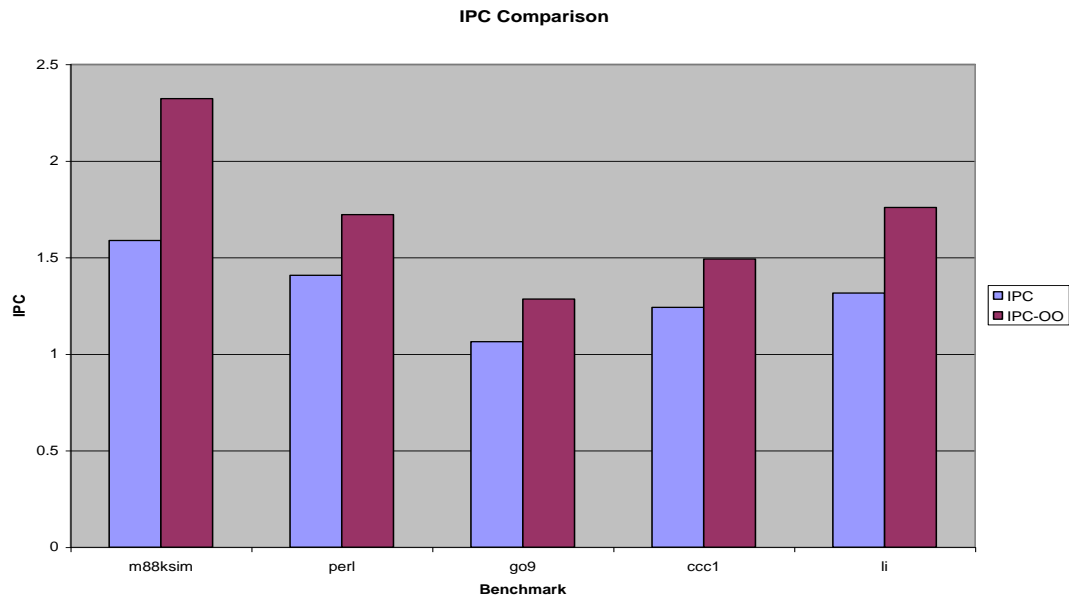
Percentage of the cycles that nothing issues from that particular FIFO.

Benchmark	NI[0]	NI[1]	NI[2]	NI[3]
m88ksim	56.52324747	56.61611205	61.66100365	60.58428264
Perl	52.0260682	55.34722511	61.75009466	66.6138204
go9	53.53914004	59.68401568	67.06457287	74.91794822
cccl	54.67163512	59.81133774	65.21609341	72.78362665
Li	51.30650659	59.1585329	63.39727395	70.30440755

5. Comparison with a 4 way out-of-order, superscalar processor

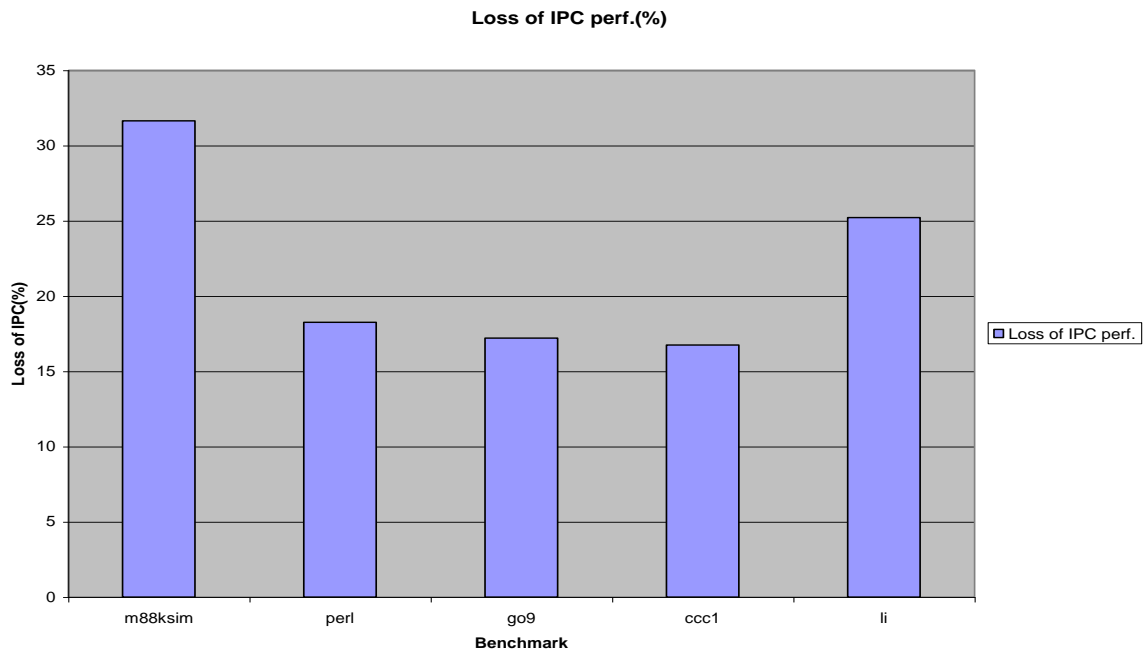
The following table has the numbers from a 4-wide, out-of-order superscalar processor.

Benchmark	Instructions Committed	Simulation Cycles	IPC-OO
m88ksim	108850300	46874764	2.3222
perl	41427374	24058139	1.722
go9	132970003	103507558	1.2846
ccc1	264897865	177617148	1.4914
li	956867545	543981664	1.759



5.1 The following table had the percentage, loss of performance of IPC of the Single-FIFO architecture when compared to the four-way, out-of-order, superscalar processor.

Benchmark	% Loss of IPC perf.
m88ksim	31.6381018
perl	18.25783972
go9	17.20379885
ccc1	16.74265791
li	25.21318931



6. Future Work

The parameters steering algorithms look to optimize are:

Inter-Cluster communications.

Number of register values to be communicated between clusters at any given time.

Inter-Cluster communications: These are achieved by wires that run between the clusters. These are potential bottlenecks in the design of the processor because wire delays are going to be significantly important in future architectures [PJS]. We can reduce inter-cluster communications by steering instructions intelligently. If data-dependent instructions are steered to the same FIFO, there is a significant drop of the communications between clusters because what is produced by an instruction in a particular FIFO is used by a subsequent instruction which has also been steered to that same FIFO. Thus, future work in this aspect would be looking at how to steer instructions better.

Number of registers values to be communicated between clusters: Each cluster primarily has a duplicate copy of the register file. By breaking up the processor into a number of smaller processing elements, we are speeding up the computation but that still requires that the register values be current in all the copies of the register file. This can contribute to quite a bit of the traffic between clusters. To reduce this, if we could identify 'dead values' in a stream of instructions and steer all the intermediate instructions using that value to the same cluster/FIFO, we could reduce a lot of that traffic.

Decoupled architectures had been proposed [JES] to help hide memory latencies and also make certain applications run faster. It would be interesting to study a similar paradigm here. There could potentially be one single FIFO which accomodates only loads and stores. While this will not eliminate communications (since the loads may themselves be creators or consumers of certain registers), the memory port requirements now fall to one or two and therefore this would be an interesting study.

7. Conclusion

The aim of this study was to design and verify the performance of very simple clusters with in order issue. The 'simple clusters' that were designed basically had one FIFO in them and so the issue-width for each cluster was one. The instructions were steered to these clusters / FIFO's using a data-dependence based heuristic. There are three classes of instruction steering algorithms. There is the functional unit dependence based steering algorithm. Then there is the control-dependence based steering algorithm and then finally there is the data-dependence based steering algorithm. The data-dependence based steering algorithms have been shown to have the highest performance and were therefore used in this study. The IPC of the single FIFO architecture implemented in this study showed a degradation of a maximum of 32% in one of the benchmarks and the others were between 16%-25%, when compared a 4-way,out-of-order superscalar processor. Around 7%-12% of the total instructions were found to communicate register values with the other clusters. The performance degradation in the IPC would be compensated for by the improvement in clock cycle time and therefore, these architectures might be implemented with reasonable success in future.

8. References

- [Pal98] : S. Palacharala. Complexity-Effective Superscalar Processors. Ph. D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, 1998.
- [JERRY]:Instruction Steering Algorithms for Clustered Microarchitecture. Jen-Chih Tseng. Masters Thesis, Electrical and Computer Engineering Department, University of Wisconsin-Madison, 2001.
- [PJS]:S. Palacharala, N.P.Jouppi and J.E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 285-290, November 1995.
- [JES]: James E. Smith: Decoupled Access/Execute Computer Architectures. [25 Years ISCA: Retrospectives and Reprints 1998](#): 231-238
- [KF]:PEW's: A Decentralized Dynamic Scheduler for ILP Processing. *Proceedings of International Conference on Parallel Processing (ICPP)*, Vol. I, pp. 239-246, 1996.

Table Of Contents

.1. Abstract	(1)
.2. Introduction	(2)
2.1 Historical Perspective	(2)
.3. Heuristic	(3)
3.1 Design	(3)
3.2 Notes on the Algorithm	(3)
3.3 Verification	(4)
.4. Results.....	(7)
4.1 Critical Delays	(7)
4.2 Inter-Cluster Communication	(8)
4.3 Not At Head of FIFO Histogram	(10)
4.4 Histogram of Full FIFOs	(11)
4.5 FIFO empty	(12)
4.6 Issue Percentages	(12)
4.7 FIFO Full and Not Issue	(13)
.5. Comparison with a 4-way,out-of-order superscalar machine	(15)
5.1 Loss of IPC performance	(16)
.6. Future Work	(17)
.7. Conclusion	(18)
.8. References	(19)