Georgia Institute of Technology

Schools of Computer Science and Electrical &Computer Engineering

CS 4290/6290, ECE 4100/6100: Fall 2013 (Conte)

Project 2: Tomasulo Algorithm Pipelined Processor

Abhishek Jain

GTID#: 902985939

ECE

Contents

1) AIM OF THE PROJECT

In this project, we were asked to construct a simulator for an out-of-order superscalar processor with a Reorder Buffer (ROB) that uses the Tomasulo algorithm and fetches N instructions per cycle. Then we were asked to use our simulator to find the appropriate number of function units and fetch rate for each benchmark.

The specifications of the **instructions** in the trace files were as follows

The input traces will be given in the form:

<address> <function unit type> <dest reg #> <src1 reg #> <src2 reg#>

Where

<address> was the address of the instruction (in hex)
<function unit type> was either "---1", "0", "1" or "2"
<dest reg #>, <src1 reg#> and <src2 reg #> are integers in the range [0..31] and if any reg number was is -1, then there was no register for that part of the instruction.

The parameters of the **processor** we were asked to simulate were as follows

- F – Fetch rate (instructions per cycle)
- M – Schedule queue multiplier
- J – Number of k0 FUs
- K – Number of k1 FUs
- L – Number of k2 FUs
- R – Number of ROB entries
- trace_file – Path name to the trace file

where all the above parameters were specified as command line parameters.

The command used to execute a single run of the simulator with user specified parameters is

./proc_sim –f F –m M –j J –k K –l L –rR < trace_file

2)THE TOMASULO ALGORITHM

2.1) Why is it needed?

The order in which the instructions comprising a program are to be executed is normally assumed to be given by the order in which the instructions are held in program storage and by the sequencing control indicated by transfer and conditional transfer instructions. However a programmer, or compiler, can produce many different but equivalent versions of a program merely by making minor alterations to the sequence in which instructions are placed. Normally the actual choice among these alternative sequences will be somewhat arbitrary, though careful programming or compilation often involves an attempt to design a program whose detailed sequences are tailored to make best use of a computer's control and functional capabilities. This can be particularly worthwhile for computers whose internal organization has been designed to attempt to overlap the use of its various functional capabilities or functional units (pipelined processors).

If really effective use is to be made of the internal capabilities of such a computer, careful attention must be paid to the detailed sequencing of instructions in frequently' executed portions of a program. This 'scheduling' can be done by an ambitious optimizing compiler, or an extremely conscientious hand-coder. There is often, however, a difficulty in achieving really optimum sequencing by such means -- that of the effects of interference (RAW, WAR and WAW dependencies) between the instructions, which if present will result in errors on executing the program.Thus there is often cause to consider the possibility of supple-menting (or even replacing) the static scheduling performed by coder or compiler by dynamic scheduling performed by the computer as it executes a program. The TOMASULO algorithm is one such algorithm which implements dynamic scheduling.

2.2) How does it work?

Tomasulo's Algorithm controls the operation of the Common Data Bus (CDB) by means of a tag mechanism. A tag used to identify separately each of the functional units which can feed the CDB. There are Reservation Stations associated with each type of functional unit. Tags are associated with each of the Registers in the register file, with the Source and Sink registers of each of the Reservation Stations. There is also a busy bit associated with each of the Registers. This bit is set whenever an instruction is issued designating that register as a sink and re-set when a result is returned to the register.

Before an instruction is issued, the busy bit of each of the specified registers in the instruction is checked. If the busy bit is zero, the contents of the register are sent to the selected Reservation Station; if the busy bit is set to one, the current value of the corresponding tag register is sent instead. The busy bit of the designated Sink register is then set to 1, and the tag number of the selected Reservation Station is entered into its tag register. Thus the tag register of a busy register identifies the last unit (in proper program sequence) which will produce a result for it.

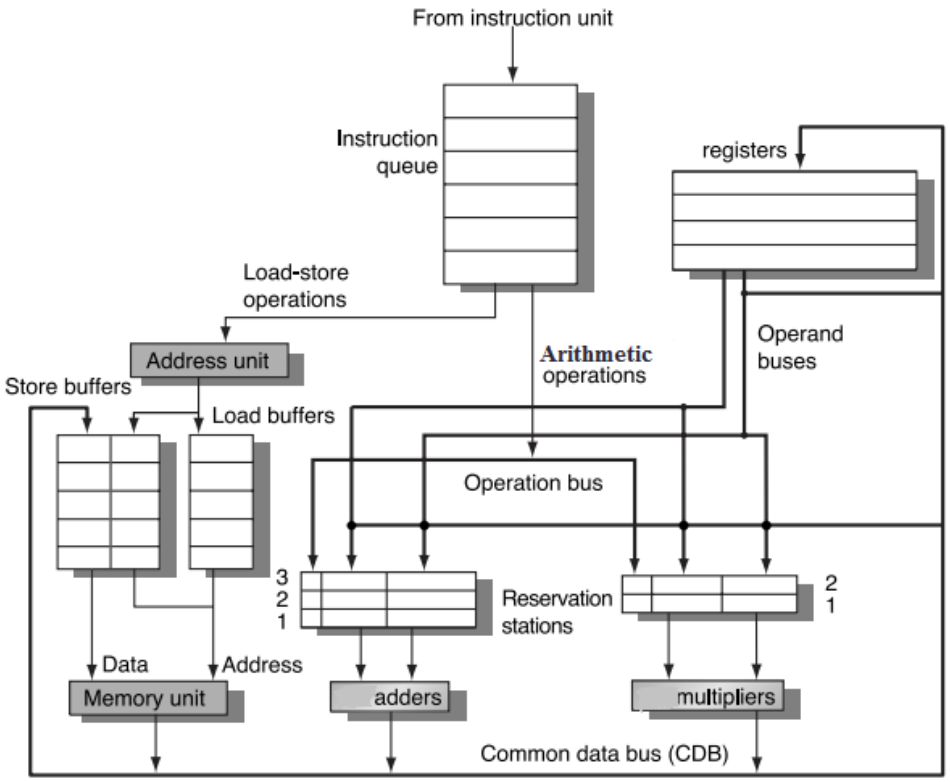
Whenever a result appears on the CDB, the tag corresponding to its Reservation Station is broadcast to all destinations. Each active Reservation Station (selected but awaiting a register

operand) compares its Sink and Source tags with the CDB tag. If a match occurs, the Reservation Station takes the data from the CDB. In a similar manner, the CDB tag is compared with the contents of the tag registers associated with the Registers. All busy registers with tags matching that on the CDB are set to the value on the CDB and their busy bits re-set.

Issuing an instruction in this system only requires that a Reservation Station be available for whichever execution unit is required. If a source register is awaiting the result of a previously issued, but as yet uncompleted instruction, or if a register is awaiting for an operand the tag associated with that register is transmitted instead to the Reservation Station, which then waits for that tag to appear at its input. Thus it is the Reservation Stations which do the waiting for operands, rather than the execution circuitry, which is free to be engaged by whichever Reservation Station fills first. Execution of an instruction starts when a Reservation Station has received both operands.

2.3) Hardware structure and key features

Hardware structure



Key features

- reservation stations
- forwarding to eliminate RAW hazards
- register renaming to eliminate WAR & WAW hazards
- deciding which instruction to execute next
- common data bus

3) IMPLEMNTING THE ALGORITHM

3.1) Approach used

The approach used was simple. Since it was evident from the trace file outputs that we had to show what happened with each instruction in which cycle, I created a sort of database in which each instruction was stored till it was finally retired. The database had fields for each of the different stages that the instruction would go through. The cycle number in which an instruction would reach a particular stage, was assigned to the corresponding field for that instruction. This way I could keep a track of what happened and when it happened for each instruction.

This approach also simplified the modelling of the scheduling, ROB and dispatch queues, as only instruction numbers had to be pushed in each queue. Each stage could then assign the correct cycle count to the instruction as soon as an instruction spent a cycle in that particular stage by looking up the instruction number in its queue in the instruction database. Also to check whether a new instruction could be brought into a particular stage, all that stage had to do was check the instruction database for instructions who had completed their work in the previous stage. As a result of this, there was no need to explicitly model each queue individually and only instruction numbers sufficed.

However, the one drawback that this approach had was that as the number of instructions would increase, there would be a significant increase in lookup time for each stage, thus causing the program to slow down. To overcome this problem, each instruction was deleted from the instruction database as soon as it was retired.

The functions used in my implementation and the tasks they perform are discussed in the subsequent sections.

3.2) The Fetch function

This function calls the read instruction function to read instructions from the trace files. The number of times this function is called depends on the number of entries in the dispatch queue. The maximum number of calls to this function is determined by the fetch width. Once the number of instructions to be fetched is decided, they are read from the trace file and then pushed in the dispatch queue. Also each instruction read from the trace file is assigned the corresponding cycle count for its instruction fetch field.

3.3) The Dispatch function

In this function an instruction's dispatch cycle field is assigned the current cycle count. First it is ensured that the Rob has some space to accommodate a new instruction. Then the particular instruction's functional unit number is checked to find out which scheduling queue the particular instruction would go to. If there is space in the scheduling queue the following would happen. Each instruction's source registers are checked, if they are not being used by any other instruction the ready fields for the source register of that particular instruction are made 1, else they are made 0 and the tag fields of that instructions source registers are filled by looking up the tags in the register file. The destination register is assigned a tag which is equal to the instruction number. This tag is also put in the register file for that particular register and the ready bit for that particular register in the register file is made 0, indicating that the particular register is in use.

3.4) The Scheduling function

In this, all instructions whose source registers are ready (which is found out by checking the source register's ready bits for that particular instruction) are marked to fire, and the instruction's schedule cycle field is given the proper cycle count. Also, each scheduling queue is made to listen to the CDB broadcast and updates the source and destination registers ready bits accordingly. Each instruction's scheduling field is assigned the correct cycle count as soon as it enters the scheduling stage.

3.5) The Execution function

First instructions in the instruction database that are ready to be execute or are marked ready to fire are found and then executed. Since, this is a simulator, execution is actually incrementing the number of cycles an instruction spends in the execution stage. For each type of functional unit, since there is a maximum number of cycles an instruction would spend in the execution unit, the execution completes as soon as that max number of cycles is encountered. Once this happens, the tag of the destination register (or the instruction number) is put on the CDB. Also the ready bit for that particular tag's register in the register file is made 1. Also in this stage, like the previous stages, the instruction's execution field is assigned the correct cycle count as soon as it enters the execution stage.

3.6) The State update function

First the instruction's schedule queue field is assigned the correct cycle number. Then the corresponding instruction's instruction number in the schedule queue is made 0, indicating that the instruction can be deleted from the schedule queue in the next cycle. Also, this

instruction is marked as completed so that the Rob function may take steps to remove it from the Rob queue.

3.7) The Retire function

It checks the instruction database for instructions that have completed and makes the instruction number of that particular instruction that has completed in the ROB 0 so that it can be deleted. However the number of instructions it decides which can be deleted depends on the fetch width. It only marks contiguous sets of instructions in the Rob as ready to be deleted. The maximum set size is the fetch width. Also, those instructions that are made ready to be deleted from the Rob in this function are assigned the proper cycle count for their retirement field in the instruction database.

3.8) The Rob and Schedule queue clean up function

These functions check the Rob and the scheduling queues respectively. Those entries which are 0 are then removed from the queues thus making more space available in the queues.

3.9) Debugging steps

Once all these functions were ready, it was time to debug the program. This was done by running the program for each trace file and putting the output in test output files. After each run I compared my output with the output provided using the command

diff –b outputfilename testfilename

By using this utility I was able to find out where I went wrong and in which cycle. Doing this several times and analyzing what went wrong each time, I was able to rectify my errors and match the outputs to the given outputs perfectly.

PS- had to use diff with –b because in spite of inserting tab spaces properly in my output I couldn’t get the spacing done correctly between entries of the same row.

4) Experimentation and results

4.1) Method of carrying out experiments

Since we had to vary the parameters as given below

- Number of FUs of each type (k0, k1, k2) of either 1 or 2 or 3 units each.
- Schedule queue sizes (m*k0+m*k1+m*k2), where m = 2 or 4 or 8
- Fetch rate, N = 2 or 4 or 8
- ROB size, R = 8 or 32 or 128

And it would have been tough varying each parameter individually by typing it in the command line again and again, a script was used to obtain the results for each file which is as follows.

```
for((F=2;F<=8;F*=2))

do

for((K0=1;K0<=3;K0++))

do for((K1=1;K1<=3;K1++))

do for((K2=1;K2<=3;K2++))

do for((M=2;M<=8;M*=2))

do for((R=8;R<=128;R*=4))

do

chmod 755 procsim
./procsim -r $R -j $K0 -k $K1 -l $K2 -f $F -m $M <traces/tracename.trace
done
done
done
done
done
done
```

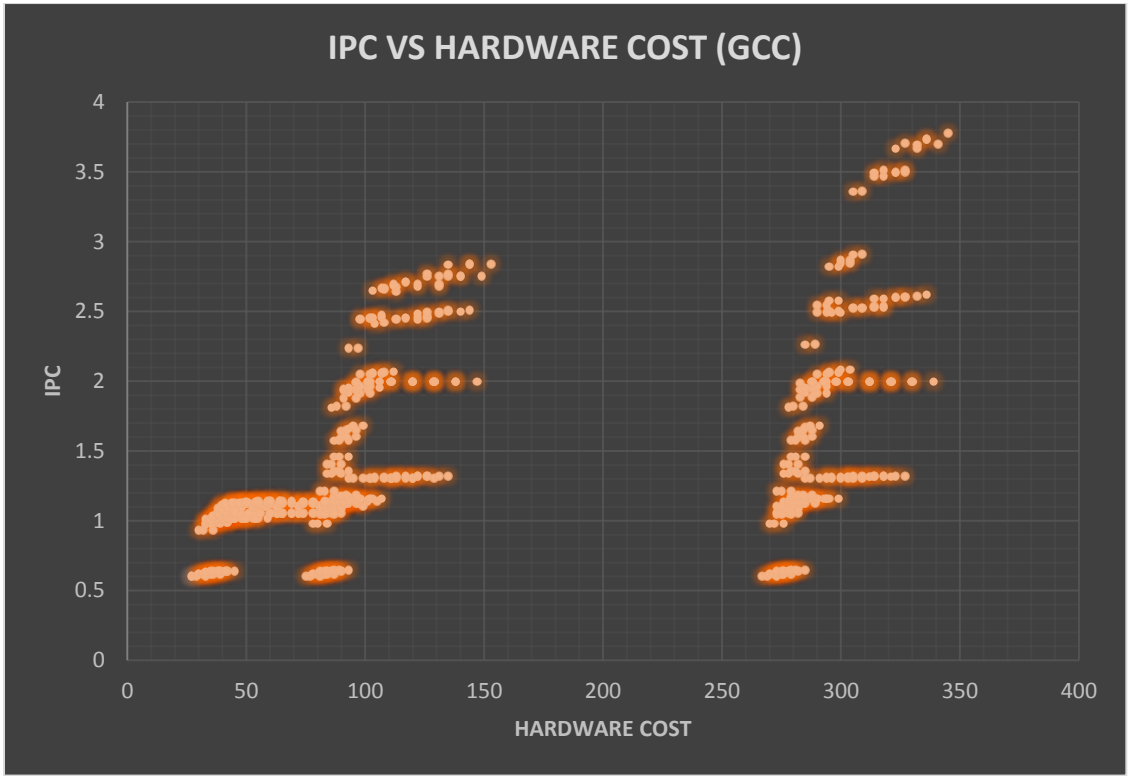
This would vary the command line parameters as requested above and give us the corresponding output for each run of a particular trace file. This was then used for each trace file and the outputs for different combinations of parameters for each trace file were saved in the corresponding excel sheet. The excel sheet of each trace file was used to plot a graph which had the cost on x-axis and the IPC on y-axis. Using the graph as a reference for each trace file, I could then find out least amount of hardware (as measured by total number of FUs (k0+k1+k2), queue entries (m*k0+m*k1+m*k2), and ROB size*2, because the dispatch queue is of the same length) for each benchmark trace that provided nearly the highest value for retired IPC.

The question however is how close do we wish to be to the maximum IPC for a particular benchmark.Since there is no concrete answer, based on the experiments carried out and the graphs, a table has been formulated in order to cater for specific situations for each trace file. In each table delta has been indicated to show the deviation from the actual highest IPC.

PS- all excel files are provided in the submission package and all entries have been arranged in order of increasing IPC from top to bottom.

Formula for calculation of cost – F+((m+1)(k0+k1+k2)) + 2*R

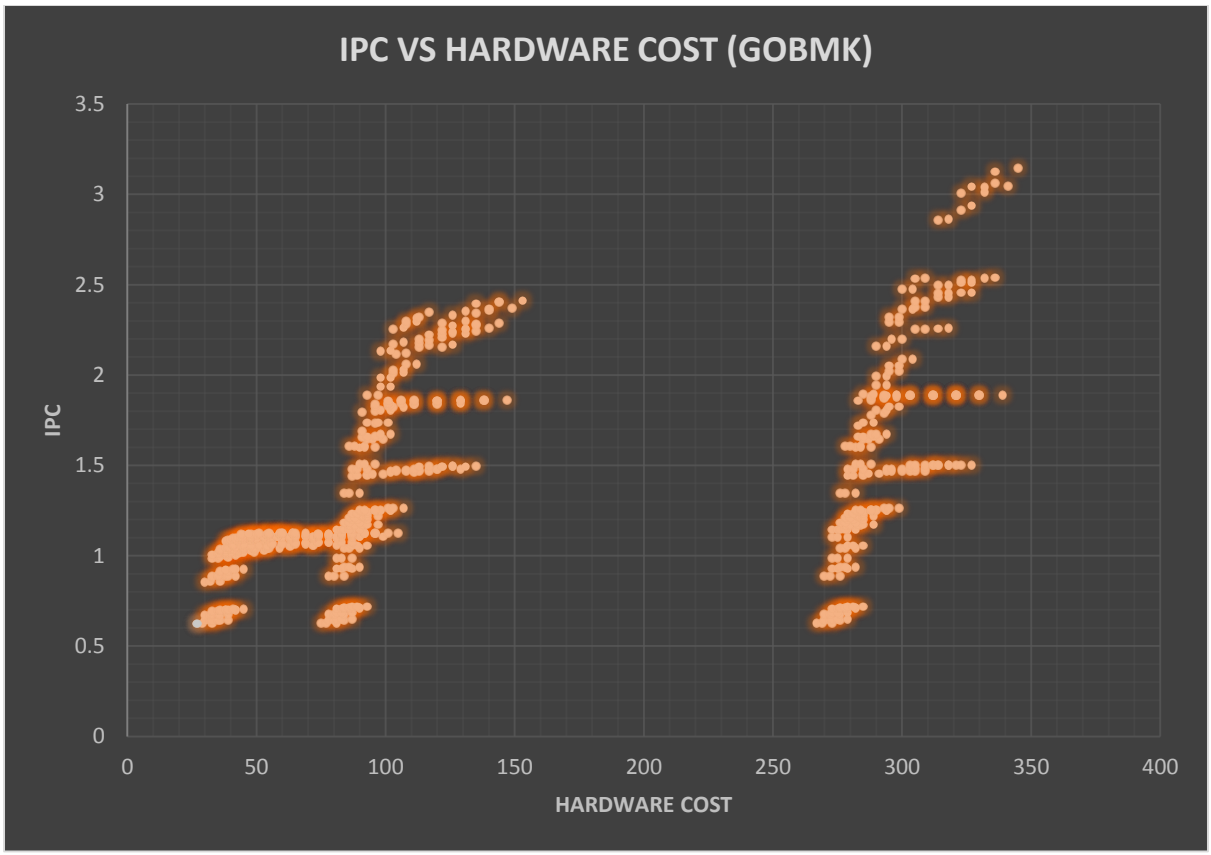
4.2) Gcc experiments and conclusion



Highest IPC – 3.77743

K0	K1	K2	F	M	R	IPC	COST	DELTA	REASON FOR CHOSEING
3	3	3	8	8	128	3.77743	345	0	HIGHEST IPC (WHEN NO COST CONSTRAINT)
3	2	2	8	8	128	3.70439	327	0.07	VERY CLOSE TO MAXIMUM IPC AND HAS REDUCED HARDWARE COST
3	2	1	8	8	128	3.5137	318	0.264	SIGNIFCANT HARWARE COST REDUCTION BUT NOT TOO MUCH DROP IN IPC
3	3	3	8	8	32	2.84131	153	0.93299	TO REDUCE THE COST OF HARDWARE TO HALF OF THAT FOR MAXIMUM IPC
3	2	2	4	8	32	2.83487	135	0.94356	TO REMAIN IN THE - 1 RANGE OF IPC AND TO REDUCE THE COST OF HARDWARE TO MORE THAN HALF OF THAT FOR MAXIMUM IPC

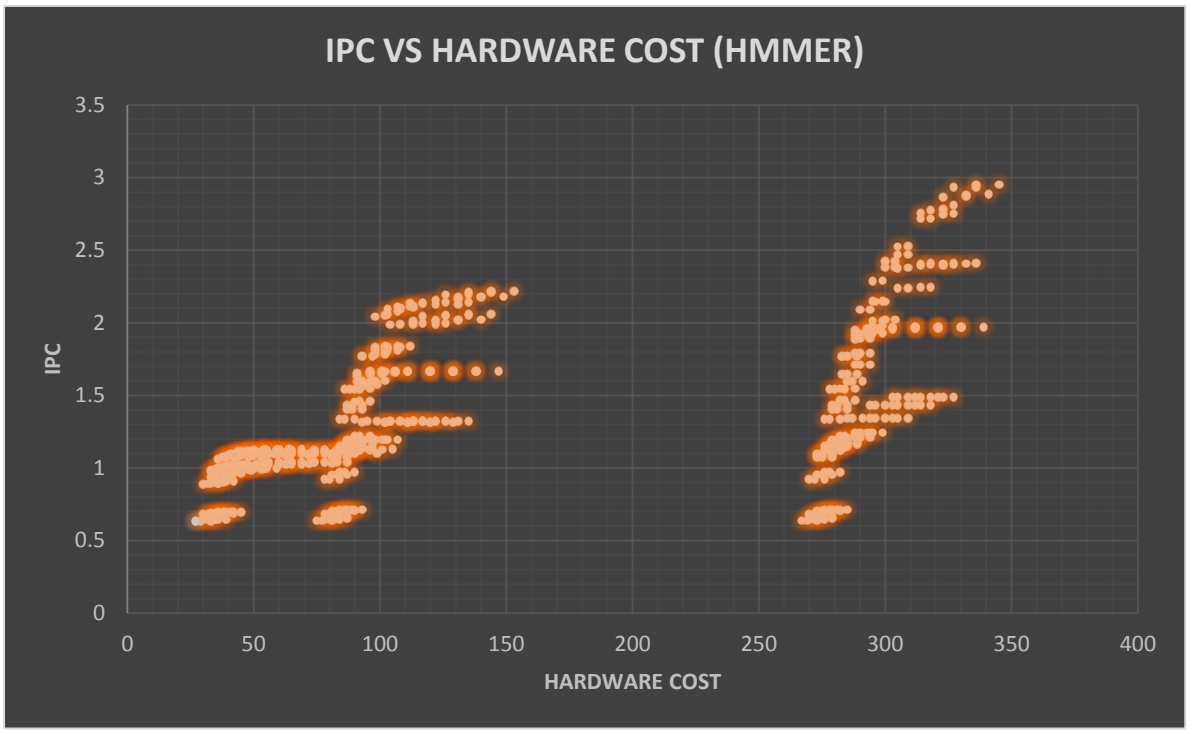
4.3) Gobmk experiments and conclusion



Highest IPC – 3.14673

K0	K1	K2	F	M	R	IPC	COST	DELTA	REASON FOR CHOSEING
3	3	3	8	8	128	3.14673	345	0	HIGHEST IPC (WHEN NO COST CONSTRAINT)
3	2	3	8	8	128	3.1251	336	0.022	VERY CLOSE TO MAXIMUM IPC AND HAS REDUCED HARDWARE COST
3	2	2	8	8	128	3.04395	327	0.1	SIGNIFCANT HARWARE COST REDUCTION BUT NOT TOO MUCH DROP IN IPC
3	3	3	8	8	32	2.41225	153	0.73448	REDUCE THE COST OF HARDWARE TO HALF OF THAT FOR MAXIMUM IPC
3	2	3	8	8	32	2.40529	144	0.74144	REDUCE THE COST EVEN FURTHER
2	2	1	4	8	32	2.15438	113	0.99235	TO REMAIN IN THE - 1 RANGE OF IPC AND TO REDUCE THE COST OF HARDWARE TO MORE THAN HALF OF THAT FOR MAXIMUM IPC

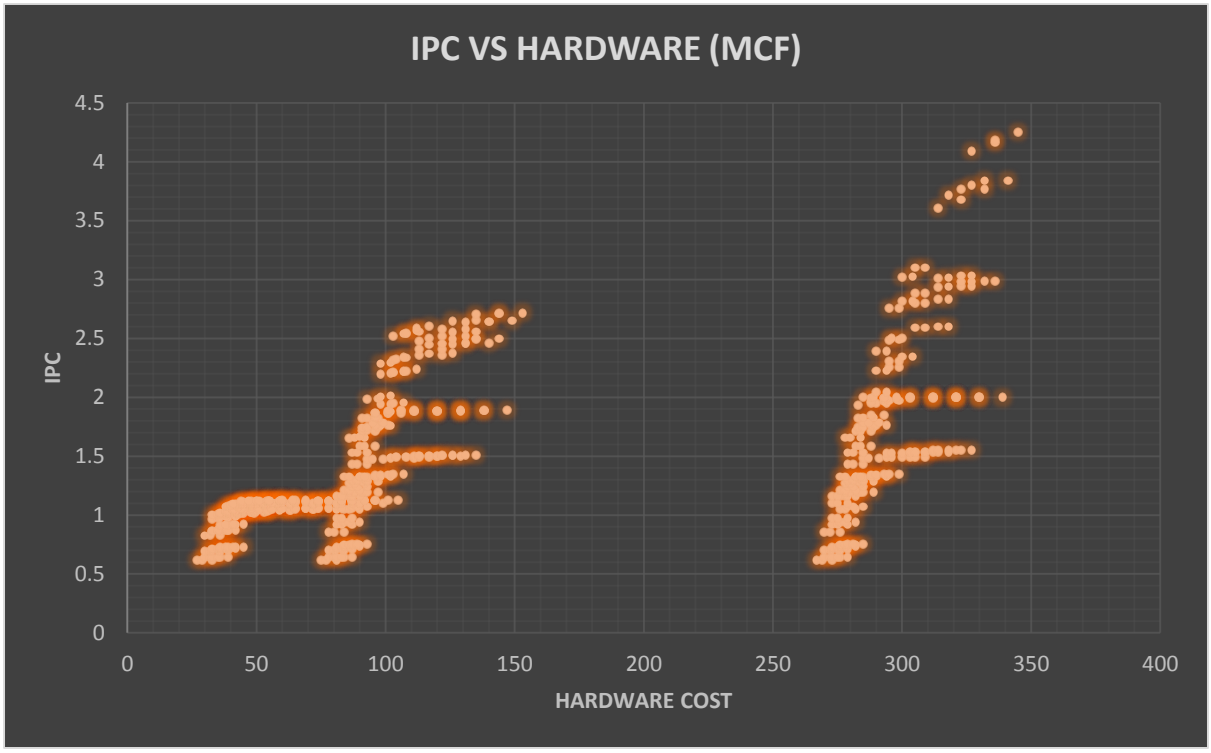
4.4) Hmmer experiments and conclusion



Highest IPC – 2.95255

K0	K1	K2	F	M	R	IPC	COST	DELTA	REASON FOR CHOSEING
3	3	3	8	8	128	2.95255	345	0	HIGHEST IPC (WHEN NO COST CONSTRAINT)
3	3	2	8	8	128	2.95125	336	0.001	VERY CLOSE TO MAXIMUM IPC AND HAS REDUCED HARDWARE COST
3	3	1	8	8	128	2.81183	327	0.14072	SIGNIFCANT HARWARE COST REDUCTION BUT NOT TOO MUCH DROP IN IPC
3	3	3	8	8	32	2.21838	153	0.73417	REDUCE THE COST OF HARDWARE TO HALF OF THAT FOR MAXIMUM IPC
3	2	2	8	4	32	2.11175	107	0.8408	REDUCE THE COST EVEN FURTHER
3	1	2	4	4	32	2.04086	98	0.91169	TO REMAIN IN THE - 1 RANGE OF IPC AND TO REDUCE THE COST OF HARDWARE TO MORE THAN HALF OF THAT FOR MAXIMUM IPC

4.5) Mcf experiments and conclusion



Highest IPC – 4.2526

K0	K1	K2	F	M	R	IPC	COST	DELTA	REASON FOR CHOSEING
3	3	3	8	8	128	4.2526	345	0	HIGHEST IPC (WHEN NO COST CONSTRAINT)
3	2	3	8	8	128	4.18305	336	0.06955	VERY CLOSE TO MAXIMUM IPC AND HAS REDUCED HARDWARE COST
3	1	2	4	8	128	3.60633	314	0.64627	TO REMAIN IN THE - 1 RANGE OF IPC AND TO REDUCE THE COST OF HARDWARE

5) Conclusion

As can be seen in the previous section, experiments were carried out for each trace file and suitable hardware configurations were found for each trace file. Also the total IPC was not allowed to go more than 1 below the highest IPC as it was felt that it would not be desirable. Also it can be seen that even though the highest IPC is achieved at the expense of maximum hardware in each case, different values of the parameters can lead to a close to optimum solution. So, with this in mind it is safe to say that there is no optimum solution for all trace files, but however it can be generalized that in majority of the cases the more the hardware, the better (the pattern for all graphs is similar when seen carefully) the IPC and there will always be tradeoffs when there are both IPC and cost constraints.

THE END