IMPLEMENTATION OF REGISTER VERSIONING

Amogh Manjunath, Raunaq NandyMajumdar

ABSTRACT

Modern superscalar processors support a large number of in-flight instructions, which requires sizeable register files. Conventional register renaming techniques allocate a new storage location, i.e. physical register, for every instruction whose destination is a logical register in order to remove false dependences. Physical registers are released in a conservative manner when the same logical register is redefined. For this reason, many cycles may happen between the last read and the release of a physical register, leading to suboptimal utilization of the register file. We have observed that for many instructions have a destination register, the produced value has only a single consumer. In this case, the RAW dependence guarantees that the producer-consumer instructions pair will be executed in program order and, hence, the same physical register can be used to store the value produced by both instructions. In this report, we discuss our implementation of a renaming technique that exploit this property to reduce the pressure on the register file. Our technique leverage physical register sharing by introducing minor changes in some other modules. We also describe how our renaming scheme supports precise exceptions. We evaluated our renaming technique on top of a modern out-of-order processor.

1. INTRODUCTION

Dynamically allocated superscalar processors exploit instruction level parallelism by re-ordering and overlapping execution of instructions in an instruction window. The size of the instruction window will depend on the factor of super scalar i.e size of the instruction window. One thing can be understood intuitively, that as in the renaming procedure of the OOO processors we use different physical registers to move around the write after write and write after read hazards, the number of physical registers needed for such operation will be dependent on the depth of the pipeline and the instruction window size, and the last thing we want to have even after making such huge progress in the field of computer architecture is structural hazards due to unavailability of sufficient amount of physical registers. This possibility of increasing size of the register file has other implications in terms of energy consumption, access time, and area. This work is motivated by the observation that in a significant amount of instruction with a destination register, the register has a single consumer, which means there is consecutive Read after Write hazards. This forces the issue queue to issue the instructions sequentially to maintain the true data dependencies in the instruction execution. Hence observing this property we can understand that the producer and consumer can use the same physical register again and again. But again some might argue that what happens when some older instruction raises an interrupt or an exception, and to maintain the precise state of the processor we need to roll back the younger instructions. I mean that was one if the motivating factors of allocating different physical register for same destination even for single use instructions. We can go around that issue by keeping versions of the same physical registers which can be implemented with shadow cells and thus consumes less power. The rename map table to this end has been extended indicating that whether the register is a single use case or not. An instruction when looks up the source register

value from the register map table when finds this bit clear then it is the first consumer of the value and there is no older instruction. To detect the absence of the younger consumers there are two cases. One which I believe is the simplest one is when the producer is the consumer and is redefining the register then it is guaranteed that the source value will only exist for that instruction only. This guarantees that there is no younger consumer of that value. If we analyse the benchmarks then this is the case for most of the instructions. In case the consumer is not the re-defining instruction then it is determined by a single use predictor as described in the principle paper and also later in the following discussion. To be able to recover the state of the processor in the case of the branch mis-prediction, interrupt or exception the paper proposes to use a multiple bank register file with check pointed register bank using shadow cells. Check pointed registers are allocated whenever it is predicted that a register might be reused. Hence the former value of the register can be recovered in event of a misprediction, interrupt or misprediction. The project report present the analysis of benchmarks that shows a high opportunity to reuse physical registers and keep versions of such registers. It also talks about register renaming technique which reduces the pressure on the register file by enabling register sharing The remainder of the paper consists of the following section. Section 2 consists of the register renaming technique described in the principle paper. Section 3 consists the discussion on our implementation of the targeted technique in our project. Section 4 consists of the results. Section 5 consists of our short comings and further improvements possible in our project followed by the conclusion appendix and glossary.

2. REGISTER VERSIONING

The work described in the paper is primarily motivated by the observation that in a significant percentage of the instructions the value stored in the register is a single use register. In conventional renaming techniques the single consumer instruction allocates new register for the destination register. However in this case as the instruction is a single use register we can store the value in the same register itself and can be reused again. This will help to reduce the pressure from the register renaming. Note that according to the paper 85% of the instructions require new physical register renaming. But we can find many instructions where the source and destination register are same.re the single use case can be used. If we consider the sample instruction set

I1	ADD R1, R2,R3
I2	ADD R1, R1,R4
I3	MUL R1, R1, R1
I4	MUL R1, R1, R3

Table 1: Sample set of instructions showing RAW, WAR, WAW hazard possibilities

For the above mentioned instruction subset if we follow conventional register renaming then we can assure that R1 will be renamed 4 times as it is destination register for all the instructions. But there are RAW hazards among the instruction subset so all the instructions will be followed sequentially. I2 cannot be issued before I1 completes, which will followed by I3, I4. If this happens we can only once rename the R1 register. For example in the conventional register renaming scheme instruction I1 renames register R1 to free physical register P1, which is used back in I2 as source register and destination R1 is renamed to P2, which is used as source registers in instruction I2 and destination register R1 is renamed to P3 which is again reused in instruction I4 and destination R1 is renamed to I4. But as already discussed due to true dependencies we cannot benefit much from register renaming we can reuse

the first physical register the instruction I1 allocates for register R1 (more discussed later). It is found that only 1.2% of instruction reuses the same physical register 3 or more times so we can provision the reuse of the same physical register for 3 time.

In the above mentioned instruction subset discussed above out of order processing is not achieved because of the RAW dependencies. Moreover the single use condition guarantees that the value is not used by any other instruction. In other words there is no other instruction reading the value between the producer and the consumer chain of the instructions that might cause WAR hazard. Hence there is no requirement of the keeping the four values alive in different physical registers for the correct program execution. The paper proposes to reuse the same physical register in the aforementioned condition, i.e when two instructions exhibit a RAW dependence and the second instruction is the only consumer of the value.

To leverage the physical register sharing the paper proposes a new hardware called physical register table (PRT). The PRT contains one entry per physical register. The table is indexed by physical register index. Each row consists of two attributes. One is the read bit and other is the count. The read bit is set 1 whenever the physical register version is being read. If the read bit is set to 1 then it indicates that the physical register is the source operand of an in-flight or a committed instruction in the pipeline. On the contrary, if the Read bit is clear, it indicates that no consumer of the value has been found yet.

When the first consumer of a register is being renamed, in order to reuse the source register for the destination register we also have to verify that there will be no future consumers. In case the consumer is also the redefining the first use register it is guaranteed that there will be no younger consumer of the value. In case the instruction being renamed is not redefining instruction, a simple single use predictor is employed to decide whether the same register is reused or a new physical register is to be allocated. The 2 bit counter keeps track of the instructions sharing the same physical register and it is used to maintain the true dependencies. Due to sharing of the same physical register, with same physical register ID, is used to identify different values produced by different instructions. As different instruction with earlier mentioned constraints use same physical register ID, it will be tough to wake up correct dependent instruction with same ID, so we have to modify the ID to wake up correct instruction. So we append the counter value with the register and it thus produces an unique ID. Every time a source register of an instruction is renamed, the Register Map table is accessed as in the conventional approach to get the physical register ID that is being assigned to it. Then it is used to index the physical register table map to get the count of that register which will indicate the version of the register. When accessing the index with the physical register table we set the read bit to 1 indicating that an inflight instruction will read from that value. For renaming destination register we need to see whether there is a chance of reuse or not. For this purpose, the renaming logic checks first the *Read bit* of the source registers in the PRT. If this bit is zero for some of the sources, it means the instruction being renamed is the first consumer of the value stored in that register. The corresponding entry in the Register Map Table is updated to map the logical register to the physical register being reused. In addition, the *Read bit* is set to zero and the *2-bit counter* is increased in the corresponding PRT entry. In case the instruction cannot be identified as the single consumer of a source register or the *2-bit counter* is saturated, a new physical register is allocated. The Register Map Table entry is updated with the ID of the allocated register, whereas the *Read bit* and the *2-bit counter* are set to zero in the PRT. In case the physical register cannot be reused, a new one is

allocated. The old register is released when the redefining instruction commits. Therefore, if a physical register can be reused the technique mimics the behaviour of a release on-rename scheme, otherwise it works as the conventional release-on-commit approach. Conventional renaming schemes stall when the list of free registers is empty. In the paper's approach, the renaming will be blocked only when there is no available physical register and there is no possibility of reusing a register. Modern out-of-order processors use dynamic speculation to issue an instruction before it is known whether or not the instruction should be executed. In our renaming scheme, multiple instructions may reuse the same physical register, and each instruction in a chain of reuses overwrites the value produced by the previous instruction. Since values in a shared physical register are speculatively overwritten, it is necessary to recover the previous value of a register in case of a branch misprediction or an exception between two instructions in the chain of reuses. In order to support precise exceptions, the different versions of a shared register must be kept. Shadow bit cells are a cost effective solution to keep previous values of a register. When an instruction is being renamed, if it needs a new physical register (i.e., the source registers cannot be reused because they do not have shadow cells available or they are not the first use), a hardware predictor determines the type of the register which should be assigned to it. Whenever a physical register is released, the entry in the register predictor table that has been used to allocate this register is updated to reflect the actual number of reuses. On the other hand, if a register that is predicted to be single-use (i.e. has been allocated in a bank with one, two or three shadow copies) is detected to be used more than once, the corresponding entry in the predictor is reset to zero. Finally, if a register predicted to be single-use is tried to be reused (i.e. it is the source operand of an instruction and it is the first use) but there are no shadow cells available, the register is not reused and the corresponding entry in the predictor is increased, so that next time it allocates a register with a greater number of shadow copies.

3. OUR IMPLEMENTATION

3.1 Physical Reg Table

As mentioned in the paper we need to create a new hardware called physical register table. It has to be indexed by the register ID that we get from the rename map table. The number of slots or index in the physical register table is equal to the number of integer and floating-point registers present in the architecture. For the gem5 out of order processor it is 256 integer registers and 256 floating point registers. There is another set of integers which is miscellaneous registers and it does not need renaming. Each index has two attributes, a read bit and a count. The read bit is set high when the register is being read which means it is a source to an inflight or committed instruction. It is zero when the register is being written to (register is source as well as destination) or hasn't been used even once or it's returned back to free list. There is a 2bit saturating counter. For every access to the index in the PRT table we are checking if the value is 4 or not. If it is such then we are changing it back to 0. The PRT module has two functions and both of them returns the version which consists of the physical register number and the count. The return type is a pair formed by standard C++ function std::pair<>. One of the functions named setPRTEntry modifies the read and count variable of the physical register index depending on the condition - when source and destination is same indicating the single use case or when they are not the same new physical register is needed. The other function is the lookup function which when called from other

classes looks up the certain location based on the physical register index value returns the correct version of that physical register. Each entry in the physical register table also contains associated max count value which is dictated by the register

predictor which tells whether the register is single use or not. It thus provides the maximum number of the shadow cells required for the same and that will be the max count for that particular register. Whenever the physical register index of the table is being accessed it is checked whether the counter is saturated or not. If not then a new version of the same register is generated. As of now for the baseline gem5 implementation we have 256 floating point and 256 integer registers and some miscellaneous registers which is not renamed. Hence, we created a 512 entry physical register table with all its initial values set to 0. The first 256 of the 512 corresponded to the physical integer registers and the next 256 corresponded to the floating-point registers. Let's take a certain instruction set and elaborate the algorithm

PC	Instr ID	Operation	Dest	Src1	Src2	Dest	Src1	Src2
1	I1	ADD	R1	R2	R3	P1.0	P2.0	P3.0
2	I2	LD	R3	M(X1)		P5.0		
3	I3	MUL	R2	R3	R4	P6.0	P5.0	P4.0
4	I4	ADD	R1	R1	R4	P1.1	P1.0	P4.0
5	I5	MUL	R1	R1	R1	P1.2	P1.1	P1.1
6	I6	MUL	R1	R1	R3	P1.3	P1.2	P5.0
7	I7	ADD	R5	R1	R2	P7.0	P1.3	P6.0
8	I8	SUB	R2	R5	R1	P6.1	P7.0	P1.3

Table 2: Sample set of instructions

The physical register table is filled up like this.

Phys_Reg	Read bit	2bit Count Value
P1	0 (1) 1(2)	0 (1) 1(4) 2(5) 3(6)
P2	0 (0) 1(1)	0(0)
P3	0 (0) 1(1)	0(0)
P4	0 (0) 1(3)	0(0)
P5	0 (2) 1(3)	0(2)
P6	0 (3) 1(8)	0 (3) 1(8)
P7	0(7)	0(7)

Table 3: Structure of Physical Register table showing entries for instructions in table1

As we see in the set of instructions that we described above there is consecutive RAW dependencies between logical register R1.

As the maximum number of the version allowed is 4 and the dependencies on R1 also leads to P1.3 from P1.0. Similar thing can

be viewed for logical register R2 in the earlier table. If we look into the second table we can see that each physical register's read bit is set to either 0 or 1 during the rename of a certain instruction mentioned in bracket along. The progress of the version count along with the instruction number is shown in the rightmost column of the table.

3.2 Rename Map

As predicted as we are changing the type of ID to be used by the architecture for register renaming we need to keep a track of the same in the rename map also to take care of the dependencies. We identified few hardware which required modifications as a result of our renaming modification. They are rename map table, scoreboard entries, history buffer, re-order buffer. The rename map is also a vector of registers indexed by architectural registers. Earlier it used to return a single physical register and now it should return a physical register and its version. The modifications were made in the rename function in the rename_map module. Before it used to store the previously allocated physical register and a newly allocated physical register as a pair and return that information. Now the entries are the physical register and its corresponding version. Now whenever the condition is met as such that a destination register is being allocated a new physical register from the freelist, its version is by default set to 0 and the pair is sent back to the caller. Here we will call the register predictor object which whenever the new physical register is called from the freelist module it stores the predicted count value in the predictor table hardware corresponding to the new physical register. This table will be accessed by the physicalRegTable module for that particular physical register to know the ceiling for its corresponding count variable. All the functions of the Rename map module that were returning physical register earlier will now return the version of the registers along with the physical register ID. For renaming source register if it is truly dependent on an inflight instruction which has not been committed yet then the rename map will be looked up for the same register and the most recent version will be fetched. After the version has been received then the physical register table will be indexed and the read bit will be set accordingly and as it is a source renaming then we don't need to modify the count value. For renaming the destination register there arrives two cases, one where either of the source registers is same as the destination register and when they are different. For the former case it is checked if the physical register allocated to the source register has any shadow cell left or not. If it is there then the destination register is not assigned a physical register form the free list and the source register is reused. The physical register table is looked up and the count value is modified and read bit is made 0. If there is no more shadow cell left then it is checked what the register predictor predicted about the number of shadow cells required by that physical register. If the prediction was wrong then the predictor and the register file handles the misprediction. When the source and destination registers are not same then a new physical register is fetched from the free list and the predictor predicts what is the maximum number of the count or shadow cells that the particular physical register should have and the physical register table is updated accordingly.

Arch Reg	Rename Version
R1	P1.0 -> P1.1 -> P1.2 ->P1.3
R2	P2.0 -> P6.0 ->P6.1
R3	P3.0 -> P5.0
R4	P4.0
R5	P7.0

Table 4: Modification in the register map highlighting the transition of the renamed registers

As the algorithm of register renaming has changed so we need to update the rename map accordingly. The Rename Version column in the earlier table shows the progress of the renaming and register versioning for a logical register

3.3 Scoreboard

The scoreboard used in the ooo processor in gem5 is a simple array of Boolean type indexed by the physical register index which either sets the index or resets it. Now as each and every register now can have maximum register then we need to increase the size of the scoreboard also, as of now. After simulation when we finalize the number of physical register taken by the modified simulation we can reduce the length of the scoreboard. The unique ID of the renamed register which contains the physical register number and the version number is used to create a unique row number of the scoreboard. Whenever the scoreboard is accessed from any of the stages with either setScoreboard, getScoreboard, or unsetScoreboard function the unique row ID derived from the physical register number and its corresponding version is passed on to the required function. Whenever a certain register is updated it is set in the scoreboard to notify the dependency graph to wake up the proper instruction. The scoreboard index can be accessed by this simple mathematics formula which linearizes the physical register number and the count value. The formula is

Scoreboard_Index = Physical_Reg_Version.first * 4+ Physical_Reg_Version.second.

Where the addend gives the physical register number and the augend gives the version number of that particular physical register.

3.4 History Buffer

The rename stage implements a history buffer which has the following entries. The instruction sequence number, architectural register, old physical register which was renamed to the architectural register, the new physical register which is used to rename the architectural register. As the renaming technique has changed so the history buffer also needs to change. Now the in place of the physical register the history buffer will contain the physical register version in place of both old physical register and new physical register. The history buffer is called when there is an interrupt or exception caused by an inflight instruction. When interrupted, precise state condition must be satisfied, which means that all instructions older than the interrupting instruction has to be committed and all instructions younger than the said instruction needs to be start afresh after the interrupt has been serviced.

For the same reason it is necessary whenever we need to squash or switch out instructions due to interrupt the newly allocated physical register to a architectural register is to be sent back to the freelist and the previous rename register is mapped to the architectural register. For the modified case if we find that the old and new renamed version's physical register part is same then we need not put the od register back into the freelist on the contrary if they are different then we need to put back the new physical register back in the freelist.

For both of the cases we need to update the count value of the corresponding physical register. If the old and new physical register is same and only vary on the count value, we need to update the count value of the said physical register in the physical register table. For history buffer specifically we need to decrement the count value of the said physical register.

Even for the case when there is different physical register for the old and new rename version we need to put back the new version's physical register to freelist and update its count value to zero in the its corresponding physical register table index. As the read bit for the same was zero already we need not update the same for the register. The following is the pseudo code for updating the history buffer

```
{
    1. If(history_buffer.previous_version.first ==history_buffer.new_version.first)
            a. Decrease the count value in the PRT table
            b. Update rename map
            c. Don't add register back to free list
    2. Else
            a. Decrease the count of the new register
            b. Add the new register back to free list
            c. Update the rename map
}
```

Pseudo Code 1: Modifications made in the history buffer

3.5 Register File

In the baseline version there are two register tables one for integer register file and the other is floating point register table. One thing that remains unchanged for the register renaming technique is we don't rename any register falling under miscellaneous register or a zero register. Now as the register renaming technique is changed and as we have versioning of the register we need to create shadow cells. Shadow cells have the property of not to directly accessed by the programmer and can be only accessed but the main register location. Shadow cells have the advantage of being low power consuming which nullifies the extra area overhead it brings in the circuit level. For the modelling purpose we increased the width of each register file four times and propose to decrease the length of the register files to decrease the pressure from the register file which will reduce the pressure. For example register P1 is an integer physical register and it can have maximum 4 versions. The register is 32 bits wide, but in our design we increased the width is now 128 bits wide where the [31:0] is for version1 and so on to [127:96] is the version4. Now if we consider the setup from the circuit perspective, as there is only one main version which is version1 and the remaining ones

being implemented using shadow cell we can use a single instance of the clock gating module for all these four versions in which most of the power will be consumed by the main version and the shadow cells take very less power which will save power when the register file is not in use or in idle state.

The following is a simple illustration of the registers implementing four different versions of the register. Where the count 0 being the main version and remaining being the shadow cells.

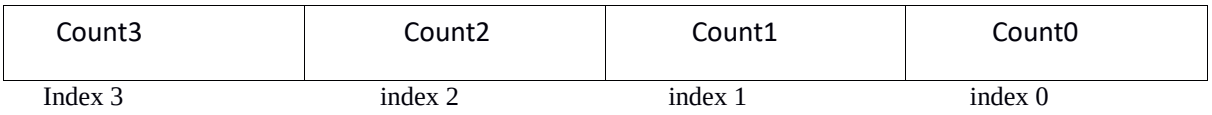


Figure 1: Layout of the register with shadow cells

3.6 Register predictor

As discussed earlier we need a predictor for the allocating the number of shadow cells in the lieu of the condition that either the source register's physical map cannot be reused due to shadow cell exhaustion or the when the destination logical and source logical registers are not same. As discussed in the paper the predictor simply uses a hashing function from the program counter and predicts the number of shadow cells required for a certain physical register. Predictor algorithm

The register predictor table is a predictor that speculates the number of shadow cell that might be used by the physical register that is renamed. This predictor is called when a physical register is renamed. As mentioned earlier, a predictor table is maintained that stores the shadow cells predicted which is indexed by the physical register ID. When a physical register is renamed and when the setPredictor function is called, it first checks if there is any previous entry for that register. If there is an entry, it will allocate that value as the shadow cell to the register. On the contrary, a simple hashing function is performed with the PC of that inflight instruction. The output of the hashing function is provided as the number of shadow cells (0, 1, 2 or 3) to be allocated to the physical register. This value is also recorded as an entry in the predictor table which will be used as a check when the particular register is used again for renaming.

The function updatePredictor is called when the instruction is committed. Difference between the shadow cells allotted and the number of dependencies for a physical register is calculated. The entry in the predictor table will be increased or decreased based on the sign of difference and the subrtracted value (‘-’ implies more shadow cells allotted, ‘+’ fewer shadow cells allotted).

3.6.1 Hashing function algorithm

For the implementation, the algorithm takes in PC as the input and gives a number 0, 1, 2 or 3 which is a 2bit number when realized on the hardware. First 16 bits of the PC are considered for computation. Nibbles are extracted and added and further a decision is performed based on a threshold to provide a 2bit number.

```
r1 = PC*0xF;
r2 = PC*0xF0;
r3 = PC*0xF00;
r4 = PC*0xF000;

sum = r1+r2+r3+r4;

if(sum<300)
pred = 0;
else if(sum>=300 && sum<450)
pred = 1;
else if(sum<=450 && sum<600)
pred = 2;
else
pred =3;
```

Pseudo Code 2: Prediction algorithm

With the following threshold at each stage, we found a considerable improvement in the hit rate of prediction.

The predictor values are updated at the commit stage if it is found there was an underutilization or over utilization of the shadow cells for the physical register. So, when that particular register is renamed, by the bookkeeping, it can be allotted the updated number of shadow cells.

Physical Reg	Predicted Value	Actual dependencies	Hit/Miss
P1	3	3	Hit
P2	1	2	Miss
P3	1	0	Miss
P4	0	0	Hit
P5	0	0	Hit
P6	0	1	Miss
P7	1	1	Hit

Table 4: Structure of the register predictor table, showing example values for each column

```
removeFromHistory(){
    1. If(predicted_value == actual_dependency)
        a. Hit++
    2. Else
        a. Miss++
}
```

Pseudo Code 3: Logic to check if the prediction was correct or not

3.7 Rename Stage

This section summarizes the entire rename stage's modification for the new register renaming technique. We merged the source and destination register function into a single function where our aim was not to rename every destination register we come across. First the source register is checked whether it is ready or not. If its corresponding unique ID location in the scoreboard is ready then the corresponding value is 1. If it is ready then the source is ready either due to completion of the earlier instruction which was writing to the same physical register or that register is being used first time. Then the destination register is checked if it is same as the source register or not. If they are same then physical register version of the source register is fetched, which contains the physical register and the count value. If the current count value is less than the predicted count value then source registers physical version is reused and the count value is increased in the physical register table. History buffer is updated. For example, if the source register was mapped to P1 with count 1 and the predicted value of the count is 3, the destination register is mapped to P1 again and the count is increased to 2. For example, the next instruction also has the same destination register as of the earlier instruction we need to update the count value of the physical register in the physical register table. Rename map is also updated accordingly. The history buffer is updated accordingly. For example, the earlier destination register was n=mapped to P1 with count 1 and the younger destination register is mapped to P1 with count 2. Previous Physical version variable in the history buffer will hold the value P1,1 and the new Physical version variable in the history buffer will hold the value P1,2. When the instruction is committed as discussed in the earlier case when the actual prediction of the number of shadow cells matches with the actual dependency happened on that register then we get whether it is a hit or not. For the case when the source and destination register is not same, we have to fetch a new physical register from the free_list. After the free register is fetched then the predictor is called which will tell us the number of the shadow cells to be allotted for the concerned physical register. The register is updated in the rename map module and also the history buffer is updated accordingly. And also, after the instruction

has been committed then we need to check if the actual predicted shadow cells is same as the number of dependencies. After the source and destination register has been renamed properly and all of the dependent module like scoreboard, history buffer is modified we need to correctly channel the result in the correct register file. We also modified by the dynamic instruction module. The arguments are modified from simple physical register to the physical register version. The change is updated in the integer register file and floating register file. Here also like the scoreboard we generate a unique ID to index into the register file. The pseudo code described here helps in elaborating the renaming and communication algorithm.

```
Rename_source_destintion_regs (instruction, thread id){

1. Look up the physical version map of the register in the rename map = src_phy_ver
2. If src_phy_ver.second ==0
    a. Call the setPredictor () function of reg_pred_table module for that index
    b. Increase the number of new phy_reg count
3. Else
    a. Continue with the same physical register
4. Set the read bit zero by calling the setPRTEntry function of the physical register table function and don't change the corresponding count
5. Check from the scoreboard if the register version is ready or not.
6. If ready then Ok or else wait
7. Update the rename in the renameSrcRegs function which will update the appropriate register file
8. If dest_regs == src_regs[0] or dest_regs == src_regs[1]
    a. Check if there is sufficient shadow cells
    b. Reuse the same physical register
    c. Update the version in the PRT table
    d. Update the rename map
    e. Make the read bit to zero in the PRT table
    f. Update the history buffer; previous version = pair(newversion.first, newversion.second-)
    g. Unset the entry in the scoreboard
9. Else
    a. Get a new  physical register from free_list
    b. Call the predictor to predict the number of shadow cells
    c. Update the version in the PRT table
    d. Update the rename map
    e. Make the read bit to zero in the PRT table
    f. Update the history buffer
    g. Unset the entry in the scoreboard
10. Update the entry in the renameDestReg

}
```

Pseudo Code 4: Modification in the logic in the rename stage

4. RESULTS

As per the paper there is not much improvement in the CPI or IPC when the number of registers is higher than 80 in the processors. For the case where system has lesser number of physical registers say around 48 for each integer and floating point register bank then the maximum improvement in the above parameters is around 6%. The paper has managed to show improvement in the area used due to renaming technique over the area required by the conventional renaming technique and the same can be said about the power consumed by the circuit. But unfortunately those results are out of the scope for analysis for us so we present the trend in IPC or CPI we achieved along with the examples of register switching and sharing, predictor performance in this section.

First things first, we modified our system as per the specifications mentioned in the paper for better comparison. The metrics are as following;

Core	ARM ISA, 2 Ghz, 128 entry ROB; 40 entry Issue Queue, out of order processor, 3 width decoder, 3 width instruction dispatch, 48 KB, 3 way TLB, 48 enrty Fully associative L1 TLB
Caches	32 KB, L1-D Cache, 2-way 1 cycle 48KB L1-I Cache, 3 way, 1 cycle 1MB, L2 Cache, 16 way, 12 cycles 64 bytes, Cache Line size
Prefetcher	Stride Prefetcher of degree 1 2K BTB 32 Instruction Fetch Queue 15 cycle misprediction penalty

Table 5: Table showing simulation parameters

Figure 2

```
system.cpu.rename: Flattening index int  29 to 29.
global: Renamed reg 29 to physical reg 40 and count 0 old mapping was 29 and count 0
system.cpu.rename: [tid:0] New Physical register added : 40 phy count=1
```

Figure 2 above shows that whenever a register is added from the free list it is by default having count 0. here the architectural register 29 which was mapped to physical register 29 earlier is mapped to physical register 40 and phy_count variable is increased to 1 which means one new register is added.

Figure 3

```
system.cpu.rename: [tid:0] Predictor called : ShadowCell alolotted = 0 to register 40. In F
global: entering 0 1 0 case with phy reg 40 and count 1
 global: done  int lookup case with phy reg 40 and count 1
: global: Renamed reg 29 to physical reg 40 and count 1 old mapping was 29 and count 0
global: done  int lookup case with phy reg 29 and count 0
: system.cpu.rename: [tid:0]: Renaming arch reg 29 to physical reg 40.
system.cpu.rename: [tid:0]: Adding instruction to history buffer in src!==dst case(size=1),
```

Here after the register is added we call the predictor function and based on the hashing function we get that number of shadow cells allotted is 0 and as the physical register is a destination it's count is made is 1 which is the only version it should have.

Figure 4

```
system.cpu.rename: Flattening index int 29 to 29.
system.cpu.rename: [tid:0]: Looking up arch reg 29, got physical reg 40 with count 1.
global: entering 1 0 0 case with phy reg 40 and count 1
 global: done  int lookup case with phy reg 40 and count 1
: system.cpu.rename: [tid:0]: Looking up arch reg 29, got physical reg 40 with count 1.
system.cpu.rename: [tid:0]: Register 40 if ready with count 1 with scoreboard value 0.
system.cpu.rename: [tid:0]: Register 40 is not ready.
system.cpu.rename: Flattening index int  29 to 29.
global: In reg_pred.cc int Dependency set; ShadowCellNum 0 and actual dependency 1
: system.cpu.rename: [tid:0] Dependency set : ShadowCell allotted = 0 Actual dependency = 1 for register 40
global: In reg_pred.cc done int lookup case with phy reg 40 and ShadowCellnum 0
: system.cpu.rename: [tid:0] Looking up to see the number of ShadowCell : ShadowCell found = 0 for register 40
global: done  int lookup case with phy reg 40 and count 1
: system.cpu.rename: [tid:0] Cannot be reused for register 40
global: Renamed reg 29 to physical reg 41 and count 0 old mapping was 40 and count 1
system.cpu.rename: [tid:0] New Physical register added : 41 phy_count=2
```

Figure 4 it is shown that even though the next instruction has the same destination register, in this case architectural register 29 and the source also being architectural register 29, we cannot reuse the same physical register as number of shadow cells allotted

is 0 so new physical register is added. This is a misprediction of the register predictor. It should have predicted non zero shadow cells.

Figure 5

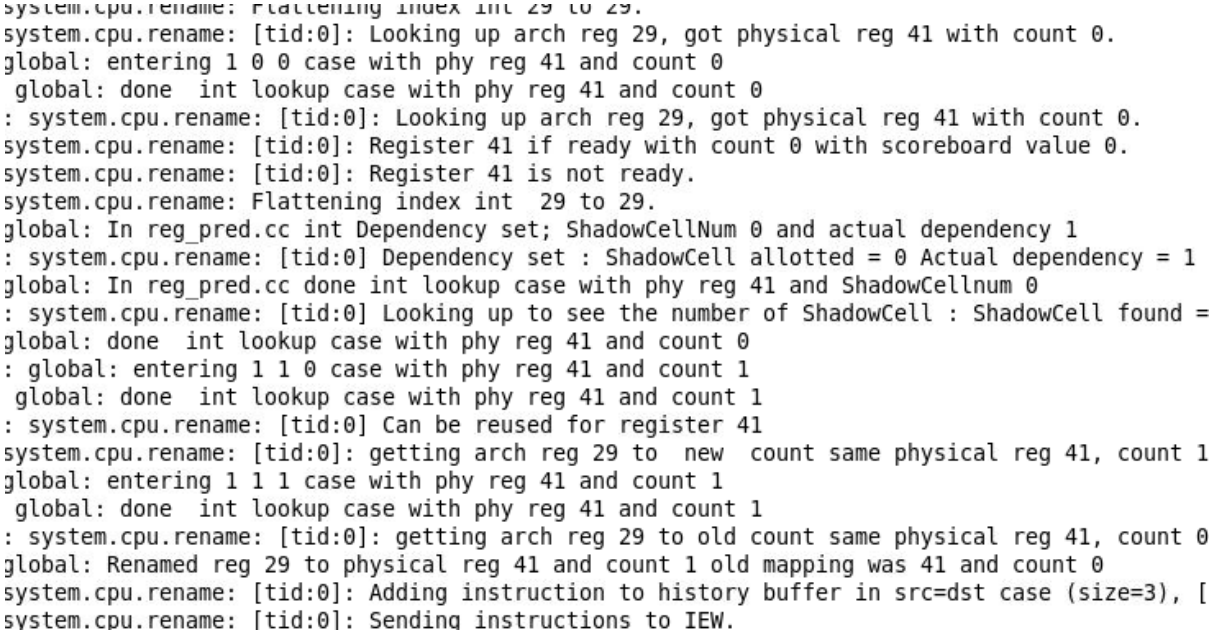
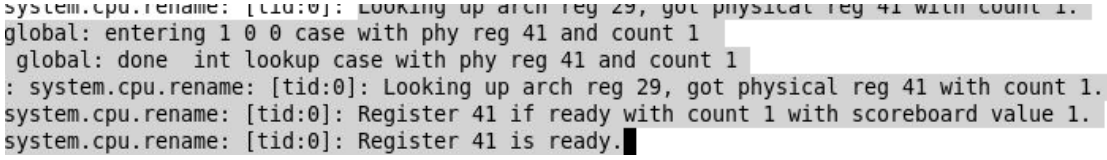


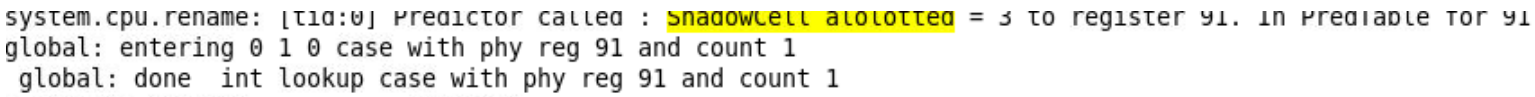
Figure 5 show next two instructions renaming. There is the case for source register is same as destination register and we can reuse the physical register 41

Figure 6



Now when this instructions have been added to instruction queue and is only ready to go when it is ready. The line in the image showing "Register 41 if ready with count 1 with scoreboard value 1" means that the particular row in the scoreboard has been set after the instruction is committed.

Figure 7



The above image shows for a case when the number of shadow cells allotted are maximum. But it does not require that much . So we again specify that we need to model the hashing function properly to get better allotment of shadow cells for better hit rate, better power utilization.

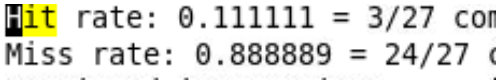


Figure 8

The above image shows a sample hit rate and miss rate. As provided by the above examples we showed that register 40 should have three shadow cells where it is given 0 shadow cells leading to mismatch between the number of shadow cells that register is

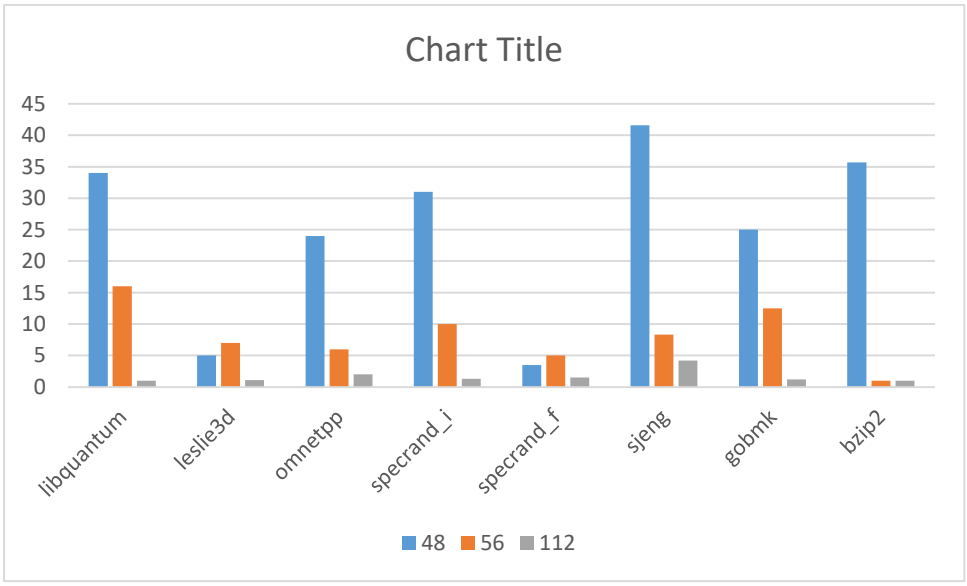
having and number of dependencies on that register. But we still get some IPC improvement but that could have been better if the prediction function is more robust.

Figure 9

```
system.cpu.rename: Flattening index int 29 to 29.
system.cpu.rename: [tid:0]: Looking up arch reg 29, got physical reg 40 with count 1.
global: entering 1 0 0 case with phy reg 40 and count 1
 global: done  int lookup case with phy reg 40 and count 1
: system.cpu.rename: [tid:0]: Looking up arch reg 29, got physical reg 40 with count 1.
system.cpu.rename: [tid:0]: Register 40 if ready with count 1 with scoreboard value 0.
system.cpu.rename: [tid:0]: Register 40 is not ready.
system.cpu.rename: Flattening index int  29 to 29.
global: In reg_pred.cc int Dependency set; ShadowCellNum 3 and actual dependency 1
: system.cpu.rename: [tid:0] Dependency set : ShadowCell allotted = 3 Actual dependency = 1 for register 40
global: In reg_pred.cc done int lookup case with phy reg 40 and ShadowCellnum 3
: system.cpu.rename: [tid:0] Looking up to see the number of ShadowCell : ShadowCell found = 3 for register 40
global: done  int lookup case with phy reg 40 and count 1
: global: entering 1 1 0 case with phy reg 40 and count 2
 global: done  int lookup case with phy reg 40 and count 2
: system.cpu.rename: [tid:0] Can be reused for register 40
system.cpu.rename: [tid:0]: getting arch reg 29 to  new  count same physical reg 40, count 2.
global: entering 1 1 1 case with phy reg 40 and count 2
 global: done  int lookup case with phy reg 40 and count 2
```

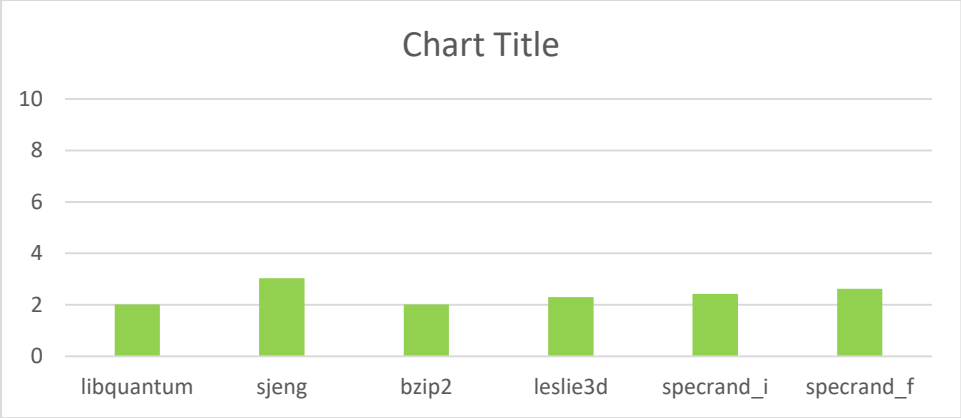
For a modified hashing function it is shown above that the number of shadow cells for physical register is allotted as 3 and it has been reused multiple times.

The screenshots above is for simulation with 256 integer and floating point registers each and the simulated for 1 billion instructions. We simulated out implementation for 1 billion instructions for certain benchmarks and we calculated the speed up over baseline implementation. We considered three cases for number of physical integers. They are 48, 56 and 112 integer and floating point registers each. The following graph summarizes the result where the y-axis shows the speedup over the base line.



Plot 1: Plots showing speed up for different benchmarks and physical registers

The following plot shows the percentage decrease in the number of renamed instructions for the case where number of physical registers in the system is 112 each for integer and floating point registers.



Plot2: Graph showing pc decrease in renamed instructions

5. SHORTCOMINGS

There are several shortcomings of our implementation. First of all the paper also compares the power and area consumed by their implementation which we were unable to compute. They specifically mention the overhead due to their new hardware like physical register table, predictor, and moreover the showstopper of the paper , the shadow cells. In our modelling the shadow cells are considered as an optional extended width of a register which might be used when needed. If we think in a circuit perspective our modelling of shadow cells resembles to selective or partial clock gated register where the flip flops of the register are enabled based on requirement. That will take less power no doubt as we need not enable all bits or all segments every time and number of registers decreases, but sharing values or transferring between different versions might corrupt values due to introduction of metastable states due to clock domain crossing as the RC value of the paths to different segments won't match due to selective clock gating scheme. Plus our register versioning technique differs slightly from the one being talked in the paper but our aim is same, to try to minimize the register renaming and save some time and power in that case. In our paper the destination register always reuses source register shadow cells if available which makes it necessary to make a robust predictor which the authors have not clarified properly. In the paper they have only mentioned about some hashing function. We tweaked it a little to share the register only if source and destination registers are same and the predictor will only tell how much shadow cells are needed.

5.1Opinion on the paper

The paper although provides a new register renaming technique but it's main aim was to reduce the power consumption of the register renaming technique but it introduces a lot of new hardware. It also shows that there is an improvement of IPC only for implementation with physical registers less than 64. Also the improvement is more pronounced in the floating point benchmarks. The improvement does not reflect for system having more physical registers. Plus it does not specify about what hashing function it specifically uses as it will decide the accuracy of the predictor. Also the other type of benchmarks that they used like Gaussian mixture models, deep neural networks did not help in speed up when the number of physical registers are high.

6. CONCLUSION

In this report we show our understanding of the principle paper on a novel register renaming technique. We describe the algorithm we followed for the implementation, the structure and the functioning of the new hardware components used in the design and also the modification done in the related stages for correct implementation. We provide the results showing successful register renaming, prediction of register shadow cells and some speedup in some benchmarks. I suggest to use this sort of register renaming technique when floating point operation is done as the results in the paper and the project shows that maximum speedup happens there only.

References

1. A novel register renaming for the out of order processors; Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, Antonio Gonzalez
2. Register Versioning: A Low-Complexity Implementation of Register Renaming in Out-of-Order Microarchitectures; Hui Zeng, Kanad Ghose, Dmitry Ponomarev
3.Register Renaming and Dynamic Speculation: An Alternative Approach: Mayan Moudgill, Keshav Pingalli, Stamatis Vassiliadis