Implementing a large, fast instruction window for tolerating cache misses using Simple scalar

Anusha Balaji
UFID: 10514929
Department of Electrical and Computer Engineering
University of Florida.
Florida-32611
abalaji@ufl.edu

ABSTRACT:

Instruction windows have a huge potential for improving the amount of parallelism in a processor. But unfortunately, increasing the instruction window size results in a trade off with the clock cycle time.

This paper implements a large instruction buffer targeted to achieve higher instruction level parallelism within a short cycle time using simple scalar. The paper is mainly designed for instructions waiting on long latency load miss instructions to get serviced. These dependent instructions are moved out of the instruction queue into a new buffer called the *Waiting Instruction Buffer*. This architecture provides better performance by effectively increasing the number of actively executing instructions without increasing the size of the cycle-critical structures. Once the long latency load miss is serviced , the dependent instructions are moved back into the issue queue for execution. Here we mainly focus on the data cache misses and the instructions dependent on those misses.

1. INTRODUCTION:

As the chasm between memory and processor increases, many modern superscalar processors use out-of-order program execution to hide memory access latencies. While instructions with long-latency memory accesses wait for their data to arrive, other ready instructions are allowed to execute, thereby maintaining a high processor utilization. In order to maintain precise interrupts, instructions that execute out-of-order must be committed to the architectural state in program order. This is especially true of instructions that change the memory state (i.e., store instructions) because it is difficult to retrieve overwritten memory values to restore architectural state.

Traditionally, microprocessor architectures deal with this problem by maintaining strict in-order execution of loads and stores. This method, though effective in maintaining precise interrupts, can lead to decreased system performance, as long-latency loads and stores at the head of the LSQ block all subsequent loads and stores from executing. Additionally, a blocking instruction at the head of the LSQ could cause the queue to saturate, stalling the dispatch of additional instructions dependent on that load.

One way to overcome this bottleneck is by using a *waiting instruction buffer* which moves the instructions dependent on long latency load miss to the buffer and then re inserts it into the issue queue after the load has been serviced. This removes pending instructions from the issue queue, freeing up entries for other instructions and allowing more instructions that do not contain data dependencies to be issued.

This design was implemented with a Waiting Instruction Buffer, which provided speedups of 10% to 65% for various SPEC benchmarks. We explore aspects of WIB design such as detecting instructions dependent on long latency operations, transferring them to WIB and re inserting them into the issue queue.

The paper is organised as follows. Section 2 provides background and motivation for this work. The simple scalar design is presented in section 3 and the evaluation of it is done in section 4. Section 5 summarizes this work and section 6 presents future directions.

2. BACKGROUND AND MOTIVATION:

Lebeck, et. al, proposed using a WIB to temporarily store long-latency instructions. This removed pending instructions from the issue queue, freeing up entries for other instructions and allowing the instructions free of data dependencies to be issued. We propose the implementation using a similar waiting instruction buffer in simple scalar to evaluate the effect of the buffer for instructions dependent on load misses.

3. IMPLEMENTATION:

We will be using modified version of simple scalar with alpha-tests and SPEC CPU 2000 benchmarks suites. The processor design will be slightly based on the alpha 21264 architecture. The various parameters for the base machine are indicated in table 1. The base configuration is used for obtaining the base simulation results.

The *waiting instruction buffer* must be implemented in such a way, that it contains and differentiates between dependent instructions of individual outstanding loads. It must allow individual instructions to be dependent on multiple outstanding loads. Every instruction dependent on a long latency load miss is allocated in the WIB. To link WIB entries to load misses we will use bit vector to indicate which WIB locations are dependent on specific load. Bit vectors are allocated when a load miss occurs. For each outstanding load miss, we store a pointer to its corresponding bit vector.

Load/Store Queue	8
Issue Queue	8
Issue Width	8
Decode Width	8
Instruction Fetch queue	8
Functional Units	Int ALU 8, Int MULT 4, FP ALU 4, FP MULT 4
Branch Prediction	Bimodal
L1 data cache	32 KB,4 way
L1 Instruction cache	32 KB, 4 way
L1 latency	2 cycles
L2 latency	10 cycles
Memory Latency	250 cycles

Table 1 base configuration

3.1 VARYING WINDOW SIZE:

I performed simulations by varying the issue window size in powers of 2 from 32 to 4096. The load store queues are kept as half the size of the instruction window. The simulations are done for the increasing size of the instruction window. Figures 1,2 and 3 show the IPC for varying window sizes and the speedup is calculated(new IPC/old IPC) for the SPEC integer and SPEC floating point and alpha-tests benchmarks.

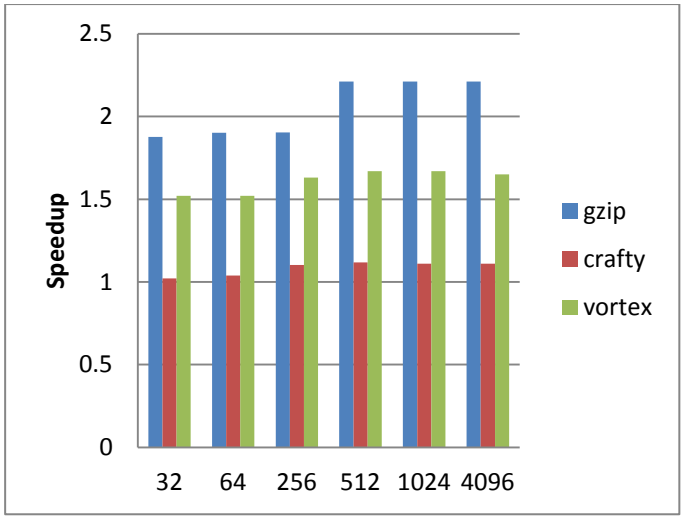

figure 1 spec2000 integer benchmarks

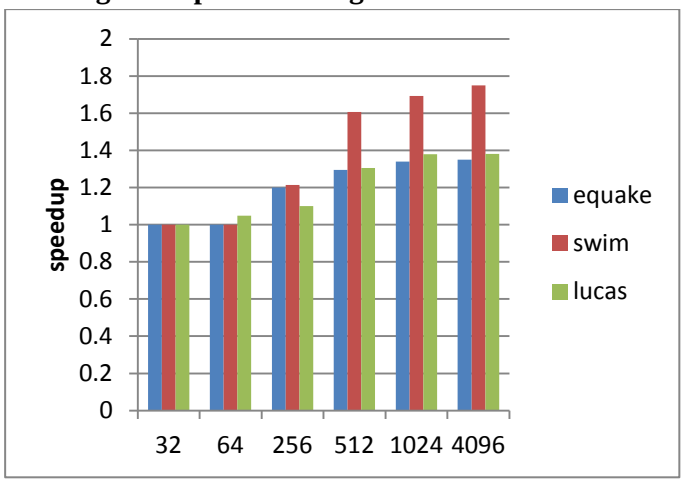

figure 2 spec2000 floating point benchmarks

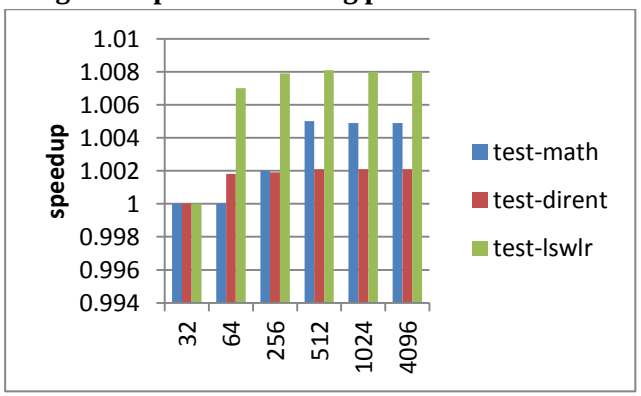

Figure 3 alpha tests benchmarks

3.2 DETECTING DEPENDENT INSTRUCTIONS:

The main objective of this paper is to divert the dependent instructions waiting on a load miss to an instruction buffer and then re insert it in the issue queue. For this, we have to identify the dependent instructions in the issue window . This is done by checking every incoming instruction in the Dispatch() stage of simple scalar code. As soon as the instruction is decoded, the dependency can be checked. This is done by using a bit vector(which is an array) to store the dependencies from a long latency load and they are cross checked against every operand of every incoming instruction in the dispatch stage. This identifies the dependent instruction and moves them into the waiting instruction

buffer rather than letting the instruction wait in the issue window and hindering the other independent instructions from being executed.

There are 2 bit vectors which are used to identify and remove the instruction from the issue window to the WIB. The operand bit vector keeps track of the names of the operand involved in the long latency load instruction and verifies it against every incoming operand. The other vector is the latency vector which keeps track of the latency period that the load instruction undergoes. The dependent instructions are delayed "latency" cycles and re inserted one by one every successive cycle based on the order in which the dependencies are serviced.

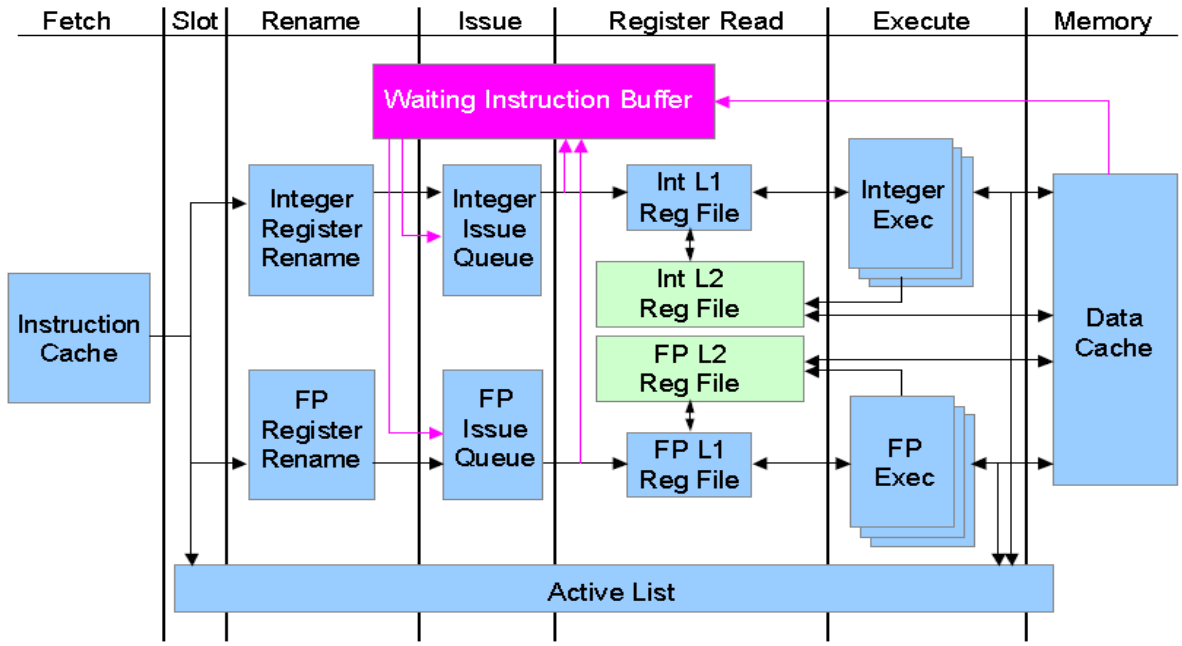

Figure 4 waiting instruction buffer

3.3 THE WAITING INSTRUCTION BUFFER:

The waiting instruction buffer contains all instructions that directly or indirectly dependent on a load miss. The WIB has been designed to identify the dependency and move them into the buffer. The waiting instruction buffer is implemented as an array. The buffer is implemented as a FIFO and they are stored inside the buffer in the program order.

The connection between the WIB entries and the load misses is done using the operand bit vector and the latency vector. When a long latency load miss occurs, the corresponding bit is set against its operand name and the latency value is stored in the latency vector. When the next instruction arrives, after decoding, its operands are checked in the vector list to find a hit. If the corresponding bit is set against its operand name, then it indicates a dependency between the load miss and the instruction. The instruction is then moved into the WIB , thus preventing the instruction from blocking the path for the other independent instructions.

Instructions from the WIB are re- inserted into the issue queue after the load miss is resolved. This is checked using the latency vector. When the latency has been serviced, the corresponding latency bit is made zero and the dependent instruction in the WIB is inserted in the issue queue. The issue queue has been designed to give priority to the instructions re-inserted from the waiting instruction buffer.

Note that some of the instructions reinserted in the issue queue by the completion of one load may be dependent on another outstanding load. The issue queue logic detects that one of the

instruction's remaining operands is unavailable, due to a load miss, in the same way it detected the first load dependence. The instruction then sets the appropriate bit in the new load's bit-vector, and is removed from the issue queue. This is a fundamental difference between the WIB and simply scaling the issue queue to larger entries. The larger queue, issues instructions only once, when all their operands are available. In contrast, our technique could move an instruction between the issue queue and waiting buffer as many times as required. Even in the worst case, when all active instructions are dependent on a single outstanding load. This requires a single bit-vector to cover the entire active list.

3.4 SQUASHING THE WIB ENTRIES:

After the WIB entries are re inserted into the issue queue, the corresponding vectors are cleared and the the waiting instruction buffer's head and tail pointers are reset so that they point to the same address.

4. EVALUATION:

The evaluation for the implementation done so far is presented by comparing it with the conventional architectural design. The simulations can reveal an increase in the performance due to the implementation of this design. The simulations are performed for the Spec2000 integer , floating point and alpha-tests benchmarks and the corresponding speed up is found.

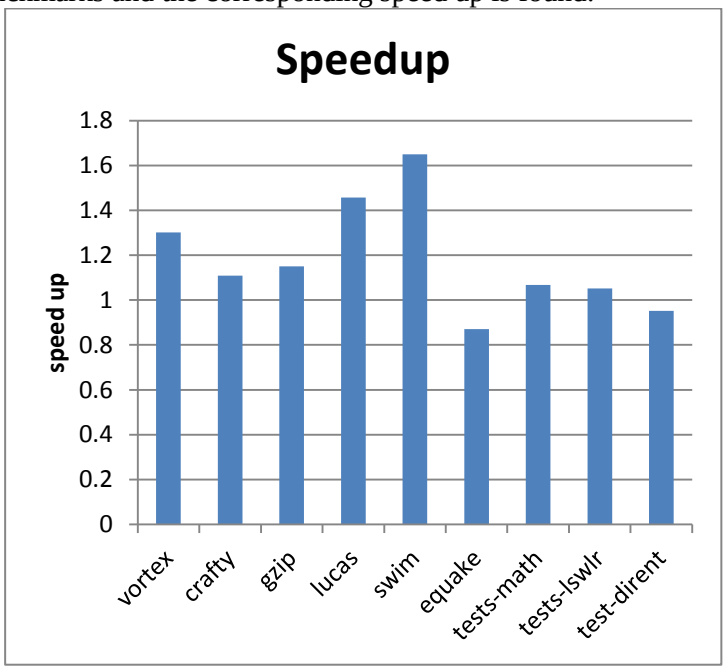

Figure 5: speedup using the WIB for various benchmarks

The overall performance speed up using the waiting instruction buffer is calculated against the IPC obtained from the base configuration. Analysing the speedup obtained, it can be seen that seven of the nine benchmarks have an improved speedup over the base configuration. The Spec2000 integer benchmarks have an average speed up of 18% and spec2000 floating point benchmarks show an average speed up of 32%. The alpha tests benchmarks show a speedup of 2.36%.

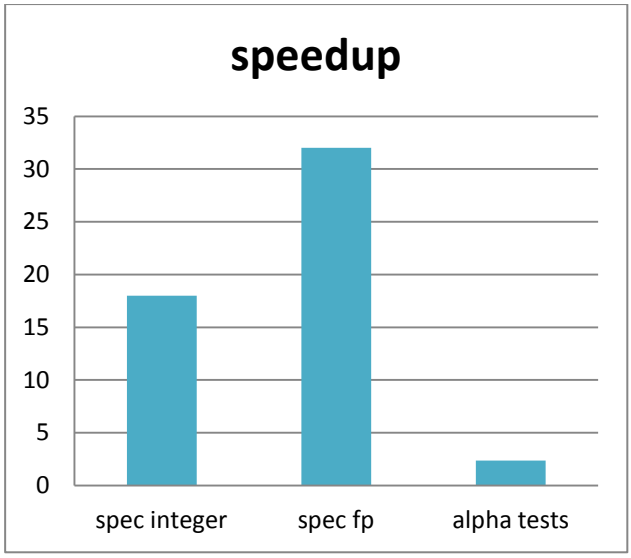

Figure 6 : average speedups of various benchmarks

We can see that for several benchmarks just increasing the issue window size produces good speedups. But, the waiting instruction buffer provides a significant improvement over the issue window size. It can be observed that the spec2000 floating point benchmarks gives the highest performance improvement. The reason for which can be stated as the higher number of long latency load misses being serviced which would contribute significantly to the improved speedups obtained.

The IPC of equake and test-dirent benchmarks show no improvement in IPC over the base configuration. This can be attributed to one of the following: the total number of load misses which cause significant blockage in the structure might be lower in this case and waiting instruction buffer is not able to perform. The other possibility is that the number of dependent instructions following the long latency load miss might be very small in order to get improvement in the IPC.

5. SUMMARY:

The main parameter for testing the overall execution time of the processor is the IPC and the clock cycle time. High clock rate can be obtained by using a small instruction window but this can limit the IPC. The implementation in simple scalar required a detailed case analysis to implement the waiting instruction buffer. The modifications were primarily done to the sim-out order.

This paper presents a new technique for achieving latency tolerance of large windows while maintaining the high clock rates. This is accomplished by implementing a waiting instruction buffer in simplescalar, which efficiently moves the dependent instructions in and out of the issue queue, thus paving the way for higher parallelism. The waiting instruction buffer successfully hides the long latency memory misses which is a primary bottleneck in most of today's processors.

6. FUTURE DIRECTION:

The waiting instruction buffer can be implemented in with many banks rather than a single buffer. This would provide more modularity to the buffer access. Further developments can be made by modifying the number of bit vectors.

Further work can include investigating the potential for executing the instructions from the WIB on a separate execution core, either conventional or grid processor.

Appendix:

Major changes done to the sim-outorder:

WIB initialisation:

```
static struct RUU_station *WIB;
int WIB_head, WIB_tail;
static int WIB_num;
static void
WIB_init(void)
{
WIB = calloc(WIB_size, sizeof(struct RUU_station));
if (!WIB)
fatal("out of virtual memory");
WIB_num = 0;
WIB_head = WIB_tail = 0;
}
```

check for instructions in WIB (whose dependencies have been serviced):

```
if((WIB_tail-
WIB_head==NULL)&&(((w_lat[in1]+WIB[WIB_head].slip)<sim_cycle)||((w_lat[in2]+WIB[WIB_h
ead].slip)<sim_cycle)))
```

fetching(pop) instructions from WIB:

```
inst=WIB[WIB_head].IR;
printf("%d",inst);
regs.regs_PC = WIB[WIB_head].PC;
pred_PC = WIB[WIB_head].next_PC;
dir_update_ptr = &(WIB[WIB_head].dir_update);
stack_recover_idx = WIB[WIB_head].stack_recover_idx;
pseq = WIB[WIB_head].ptrace_seq;
printf("slip %d\n",WIB[WIB_head].slip);
```

Dependency checking:

```
if(MD_OP_FLAGS(op)&F_MEM)
{printf("memory flag set \n");
        if(!(MD_OP_FLAGS(op)&F_STORE))
                {printf("load instruction executing \n");
                pr_v[out1]=1;reqin=out1;
                printf("%d",reqin);
                w_lat[reqin]=x_lat;}
}
else
{
        if(pr_v[in1]==1 )
                pr_v[out1]=1;
        if(pr_v[in2]==1 )
                pr_v[out1]=1;
}
```

Pushing into the WIB:

```
if( pr_v[in1]==1 || pr_v[in2]==1 )
{
        wib = &WIB[WIB_tail];
        wib->slip = sim_cycle-1;
        wib->IR = inst;
printf("wib instruction going in %d\n",wib->IR);
        wib->op = op;
        wib->PC = regs.regs_PC;
        wib->next_PC = regs.regs_NPC; wib->pred_PC = pred_PC;
        wib->in_LSQ = FALSE;
        wib->ea_comp = FALSE;
        wib->recover_inst = FALSE;
        wib->dir_update = *dir_update_ptr;
        wib->stack_recover_idx = stack_recover_idx;
        wib->spec_mode = spec_mode;
        wib->addr = addr;
                /* WIB->tag is already set */
        wib->seq = ++inst_seq;
        wib->queued = wib->issued = wib->completed = FALSE;
        wib->ptrace_seq = pseq;
        WIB_tail = (WIB_tail + 1) % WIB_size;
fetch_head = ((fetch_head+1)& (ruu_ifq_size + 1));
fetch_num++;
        break;

}
```