ABSTRACT

JAIN, SARANSH. Application Specific Parallelization. (Under the direction of Dr. Eric Rotenberg).

Traditional superscalar processors aim to exploit instruction level parallelism (ILP) by trying to find independent instructions in the program and executing them as soon as possible. Register renaming helps with removal of false dependencies. Instruction window helps find instructions that can be executed independently. The front-end is typically supported with a branch predictor which tries to predict direction of program flow. While this model works well for general code sequences, it doesn't perform well for certain applications that require huge instruction window and/or behave in a fashion that prevent effective speculation (branch-prediction, load-store disambiguation).

Post silicon micro-architecture (PSM) [11] proposes to instantiate on-the fly micro-architecture on top of general-purpose processors. On the fly micro-architecture is expected to intervene in the key-pipeline stages and help boost performance. The key merit of PSM is that using reconfigurable blocks, one can implement micro-architecture tailored to the application.

We discuss one of the directions of using PSM to improve performance: Application Specific Parallelization. We propose an implementation specifically targeting the SPEC 2006 integer benchmark astar for a wide-vector architecture. We transform one of the loops of astar to a map-reduce problem. A vector-based architecture was proposed and implemented in this work. Multiple independent threads each corresponding to an iteration of the loop are launched on the vector machine with the help of PSM. This is done despite the presence of loop-carried dependencies. However, the threads aren't squashed due to store-load violation. Instead we leverage application analysis to allow violation store-load dependencies among iterations while still ensuring functional correctness. We present a PSM based hardware reduction algorithm that targets removal of duplicate elements and thereby helps retaining functional correctness. For the baseline configuration, our implementation delivers an IPC (instructions per cycle) improvement of ~2x over a baseline superscalar configuration.

Application Specific Parallelization

by
Saransh Jain

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina
2019

APPROVED BY:

_____	_____
Dr. Eric Rotenberg Chair of Advisory Committee	Dr. Huiyang Zhou
_____	
Dr. James Tuck	

DEDICATION

To my family

BIOGRAPHY

Saransh Jain was born in Delhi, India in 1992. He received Bachelor of Engineering (B.E) from Birla Institute of Technology and Technology (BITS), Pilani in 2014. Saransh worked as a CPU verification engineer with ARM, Bangalore for about 2 years before coming to North Carolina State University (NCSU), Raleigh to pursue higher studies. During his time at NCSU, he got an opportunity to work with Intel, Hillsboro as a CPU verification intern. He is currently a graduate student under the guidance of Dr. Eric Rotenberg. Upon completion of his graduate degree, Saransh plans to join Microsoft, Raleigh as a Silicon Design Engineer.

ACKNOWLEDGMENTS

I would like to thank my parents, my sister and my brother-in-law for their continued motivation. I would like to thank them for always encouraging me to study and learn further. They have been a constant source of support and inspiration.

I am indebted to my adviser, Dr Eric Rotenberg, for his guidance and encouragement. He has a been a great mentor to me. My discussions with him have been a great source of learning for me. I also thank Dr. Rotenberg for providing ample space for independent thinking and while at the same time guiding me in the right direction.

I would like to acknowledge my committee members Dr. Huiyang Zhou and Dr. James Tuck for their willingness to support me in this work. Their teachings have helped me throughout this work.

I would also like to thank my friends, Aayush Chaudhary, Adith Vastrad and Utkarsh Mathur who have always been more than willing to help. A special thanks to Utkarsh Mathur for entertaining all my weird ideas.

This thesis was supported in part by NSF grant No. CCF-1823517, and Intel. Any opinions, findings and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the National Science Foundation or Intel.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. Introduction

1.1 Motivation

Improving IPC in traditional superscalar processors relies on finding instructions that can be executed independently. Issue-Queue (IQ) acts as a staging point from where instructions with ready operands can be issued to the execution lanes irrespective of their original program order. Out of Order (OOO) execution helps hide the latency emanating from long-latency instructions such as a load-miss. Also, a large window helps tolerate latency emanating from multiple long latency instructions. Moreover, it helps in extracting memory-level parallelism (MLP) since a cache miss doesn't stall other loads which are neither directly nor indirectly dependent on the result of the first load.

One structure that is key to high performance are the branch predictors. Branch predictors essentially try to predict the path the program will take and therefore help avoid front-end stalls. Frequency of branches in the program necessitates highly accurate branch-predictors. A standard technique to handle branch-mispredictions is to squash any younger instructions in the pipeline. While this superscalar model works well, there are some limitations:

- Providing Large Window: Applications lacking ILP need large window superscalar processors to help find independent instructions. A large window to find independent instructions translates into large IQ (and reorder buffer). There is a limit to scaling associative structures such as IQ without paying a cycle time penalty. A cache missed load instruction can create a long-data dependency chain clogging a modestly sized IQ. This may further result in stall of front-end causing significant IPC degradation. A big speculative window also requires more physical registers which is a centralized structure and may hurt cycle-time if increased beyond certain limits.
- Branch Predictor Performance: Branch predictors occasionally leverage the biased nature of program flow to predict future behavior. For this they typically target both local and global branch history patterns. However, in some cases it is difficult to predict branch direction patterns. One such case is conditional branches dependent on loads to the memory. If the changes to the load address are frequent, it is non-trivial

to predict such a branch. Furthermore, if the load that the branch depends on misses in the cache, the penalty of squashing such a mispredicted branch is huge and neutralizes the latency hiding ability of the large window [9]. Specialized branch predictors may work well for these applications, but their non-performance in other applications precludes dedicating silicon to such special predictors on the chip.

1.2 Goal

This work tries to overcome the above problems in a unique way that becomes possible with the introduction of Post silicon micro-architecture. PSM paradigm considers exploiting FPGA like reconfigurable blocks coupled tightly with the processor to instantiate application specific micro-architecture on the fly to boost IPC. We analyze and provide a sample implementation for SPEC 2006 benchmark astar which suffers from the problem of highly unpredictable branches. While different strategies can be used to target this problem with the help of PSM (like a customized branch predictor), in this work we explore a vector-based architecture that parallelizes loop iterations despite the presence of loop-carried dependencies. The proposed architecture requires simple structures that mostly work independently and require collaboration only at key stages. It avoids associative structures to allow better scaling. Coupled with replication of a non-associative single-issue width IQ, the proposed architecture can provide a huge speculative window at the cost of potentially doing redundant work. It intends to leverage the presence of a tightly coupled PSM fabric to help execute multiple loop iterations in parallel and also recover from the potential violations resulting from vectorization via reduction. The discussed work can also be thought of as a loop accelerator. The work also discusses a simple reduction algorithm that can be synthesized using reconfigurable blocks. While this work discusses an implementation specific to astar, the same can be extended to parallelize other loops since map and reduce are to be orchestrated by the PSM fabric and therefore can be programmed for an application.

Some key contributions of this work are:

- Characterization of astar and remodeling of the same to a map-reduce problem.
- Implementation of a vector architecture with a unique IQ implementation allowing decoupled execution of threads within a limited window.

- Leveraging reconfigurable hardware with the vector machine to orchestrate transformation of loops to parallel hardware threads.
- An astar specific fifo-based reduction algorithm.

For the baseline configuration of vector machine which models the fifo-based reduction, we were able to show IPC improvement of ~2x with real caches. With perfect L1 D-Cache in both superscalar and vector-based configuration, we see an even higher improvement of 3.625x.

The thesis is organized as follows. Chapter 2 discusses prior related work. In Chapter 3, we present a detailed analysis of the targeted loop in astar. In Chapter 4, we present a high-level solution which is followed by a detailed discussion of the overall operation in Chapter 5. In Chapter 6, our evaluation environment and chosen baseline configuration is discussed. In Chapter 7, we present the results. We present conclusion and future work in Chapter 8.

2. Related Work

Thread-level speculation [3,4,5,6,7] is related to the approach taken in this work. Typical thread-level speculation strategies involve mapping different tasks speculatively to different threads under the assumption that there will be no dependence violations. The threads are allowed to run independently while a monitor checks for violations. If violations are detected, the speculative work is squashed and restarted serially. In this work, while we do check for dependency violation, we do not squash speculative work. We leverage knowledge of what the program does to correct memory state.

Another related work is ReMAP[12]. It proposes to use reconfigurable fabric with a cluster of cores for accelerating and parallelizing applications. One difference is that ReMAP relies on source level or assembly level modifications for parallelization. However, in this work, we propose to use the reconfigurable fabric to intervene within the micro-architecture to help parallelize loop iterations.

3. Understanding Astar

Astar, a SPEC 2006 benchmark is a path-finding algorithm which traverses 2D graph based on estimate of cost required to reach the goal. It spends a significant fraction of its execution time in the wayobj::makebound2() function as measured by running gprof of astar natively on a linux server [9]. The function takes an input worklist and iterates over it. Each element in the input worklist contains a location in a 2D grid. Let's call the input worklist elements B[i] where 'i' is the iteration variable. In each iteration, the code looks at 8 neighbors of the element pointed to by B[i].

```
1  i32 wayobj::makebound2(i32pt bound1p, i32 bound1l, i32pt bound2p)
2  {
3    i32 bound2l;
4    i32 index,index1;
5    i32 yoffset;
6    waymapcellpt waymap;
7    i32 i;
8
9    yoffset=maply;
10   waymap=wayobj::waymap;
11
12   bound2l=0;
13   for (i=0; i<bound1l; i++)
14   {
15     index=bound1p[i];                          // access from the input worklist(B[i])
16
17     index1=index-yoffset-1;                    // computing location of 1 neighbor
18     if (waymap[index1].fillnum!=fillnum)       // Conditional branch dependent on load of form A[B[i]]
19       if (maparp[index1]==0)                   // Conditional branch dependent on load of form A[B[i]]
20       {
21         bound2p[bound2l]=index1;
22         bound2l++;
23
24         waymap[index1].fillnum=fillnum;        // fillnum update. Used to avoid duplicate calculation
25         waymap[index1].num=step;
26
27         if (index1==endindex)                  // if true indicates reached destination
28         {
29           flend=true;
30           return bound2l;
31         }
32       }
33
34     index1=index-yoffset;                      // computing location of the next neighbor
```

Figure 1: wayobj::makebound2() source code

Figure 1 shows an excerpt from the same benchmark with the computation for the first neighbor. Computation for other neighbors involves exactly the same code with updates to index1. Each of these neighbor indices are calculated using B[i] (index1 corresponds to index of first neighbor, calculated in line 17 in Figure 1) There are 2 accesses per neighbor (Line 18 and Line 19 in Figure 1 correspond to accesses for first neighbor). All these accesses are indirect

accesses of the form A[B[i]]. For the 8 neighbors per loop iteration, this translates to a total of 16 indirect accesses. Indices visited form a part of the output worklist. Once a neighbor is visited, a flag (update to fillnum) is marked at the neighbor to avoid redundant calculation in the future (line 24 in Figure 1). This flag may direct the program flow in future iterations. Majority of branch mispredictions in astar stem from the branches driven by these 16 indirect loads (82% to 97%) [10]. The reason for these mispredictions is their dependence on load. Future loads at the same PC, point to different location in the graph resulting in an unpredictable branch direction. In-fact, even future loads to the same location after the first visit changes branch direction. Therefore, it is non-trivial to correctly predict the path taken based on just PC's of the branch. Instead, one needs a special branch predictor (like EXACT [1]) to confidently predict such branches.

Cache misses are another problem emanating from indirect accesses. The input worklist provides the indices in the 2D map for each iteration. Since these indices may not follow some predictable stride pattern, accesses in the map may result in cache misses. A large window helps with hiding latency of the missed cache access and providing some level of MLP. However, these same cache misses also determine the direction of the conditional branch. Any mispredicted branch causes all the future work to be squashed. A cache miss feeding the branch makes this situation worse where the benefit of providing a huge window is diminished.

4. High-Level Solution

One possible solution to increase IPC could be to parallelize the iterations of the loop for a wide-vector micro-architecture such as a GPU. This would require privatization of induction variable for multiple iterations to run independently. Squash-free execution per iteration can be achieved by relying on execution for branch direction (stalling pipeline on detecting branch) instead of branch predictions. Parallel running independent iterations enjoy control-flow independence and can also exploit memory-level parallelism.

However, the problem with such an implementation is the presence of store-load dependencies among iterations. Figure 2 demonstrates the problem of loop-carried dependency where execution of a later iteration is dependent on the current iteration. This arises from the nature of astar which requires the output worklist to have only unique cells. The program intends to skip a node in the graph if it was visited earlier. Running multiple iterations in parallel can lead to duplicate elements in the worklist. Another problem is the address of elements in the output worklist. Since the exact number of elements that will be added to the output worklist per iteration aren't known before executing the loop, younger iterations will have to wait for the older iteration. Thus, store-load dependency between loop iterations precludes trivial parallelization of these iterations.

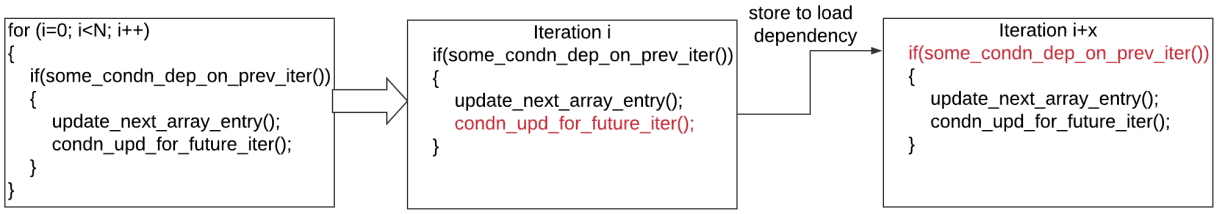

Figure 2: Store-Load Dependencies across loop iterations

4.1 Proposed Execution Paradigm

The proposed solution discussed in this work is to vectorize the iterations of the loop, ignoring the store-load dependencies. Note here, we do not recompile the binary to generate vector instructions. Instead, using PSM, we map multiple iterations of the loop to individual

hardware threads each capable of independent execution while syncing at certain stages. Since the code executed inside the loop remains the same, we can have a single fetch stream. However, one needs to identify live-ins to these threads and populate them correctly. PSM is used to help populate these live-ins. Since store-load dependencies are ignored, we can end up with redundant elements in the output worklist. We use PSM to implement a custom reduction operation operating on the sub-worklists at the end of each batch of mapped iterations. One batch of mapped iteration is referred to as a 'vector-iteration'. With respect to astar, we want to exploit the fact that the store-load violations are potential violations and not guaranteed to be always present i.e., not all work done speculatively will be wasteful or redundant. Figure 3 shows the high-level solution discussed in this work.

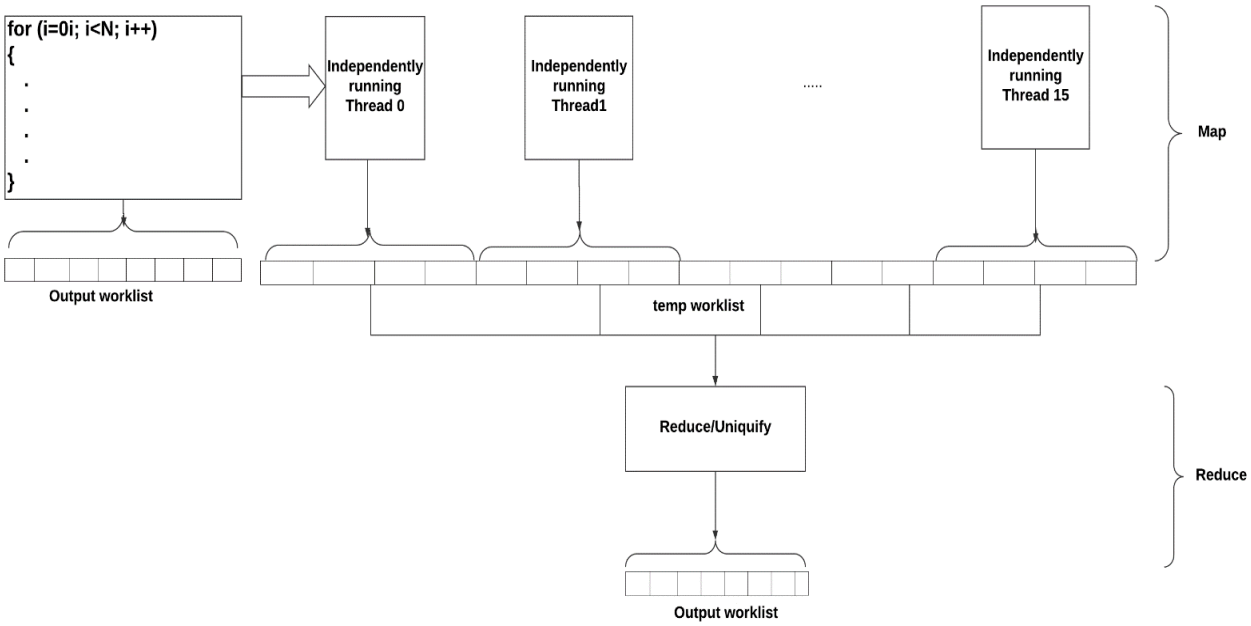

Figure 3: High-Level Map-Reduction Solution

Such a solution is very specific to astar and enabled by PSM. However, the overall idea of using PSM involves several general-purpose micro-architectures (such as superscalar, wide-vector processor) on the same silicon. Each candidate micro-architecture will have programmable hardware at key-places throughout the pipeline. The programmable hardware can be configured per application to intervene and orchestrate certain functionality. In case of astar, the target micro-architecture could be a wide-vector processor. The bitstream programming PSM fabric could encode certain PCs that help in launching parallel threads on the wide-vector

machine and also configure the PSM fabric for reduction of sub-worklists. The PSM fabric provides all this functionality 'under the hood' i.e. one doesn't need to recompile the binary.

There can be other ways to parallelize astar loop iterations while being cognizant of the store-load dependencies. One implementation could be to parallelize astar explicitly with map-reduce libraries in software. Unfortunately, a software implementation of reduce phase tends to be inefficient constituting many sequential dynamic instructions and can again suffer from cache misses and unpredictable branch direction. It might also bloat up the code size.

4.2 High-Level Implementation

As mentioned previously, the intended use of PSM involves a heterogeneous multicore chip with one or more instances of different micro-architectures targeting different types of parallelism. E.g. one possible multicore can consist of a big-window superscalar along with a wide-vector processor [11]. The superscalar can target majority applications while the wide-vector processor can target specific applications or even specific sections of the application. PSM can help orchestrate transfer of control when such sections are detected.

In this work, we discuss a wide-vector architecture targeting parallelization of loop iterations in the context of astar. Figure 4 shows the high-level implementation of the architecture that is discussed in this work.

We derive this architecture from the canonical superscalar processor and target issue concerning poor branch prediction. GPU architecture also forms inspiration for much of this architecture. Some key differences as compared to the superscalar processor are:

- Absence of branch predictors: Branch direction is determined via execution.
- Multiple IQs each bound to its own set of registers. One full set of architectural registers is available for each lane. Instruction issued/executed in a lane read/write to/from the registers of that lane.
- One universal execution unit per lane.
- In-order operation per lane
- Separate scalar and vector mode: In vector mode, one or more lanes operate, while in scalar mode, only one lane operates, and other lanes are disabled.

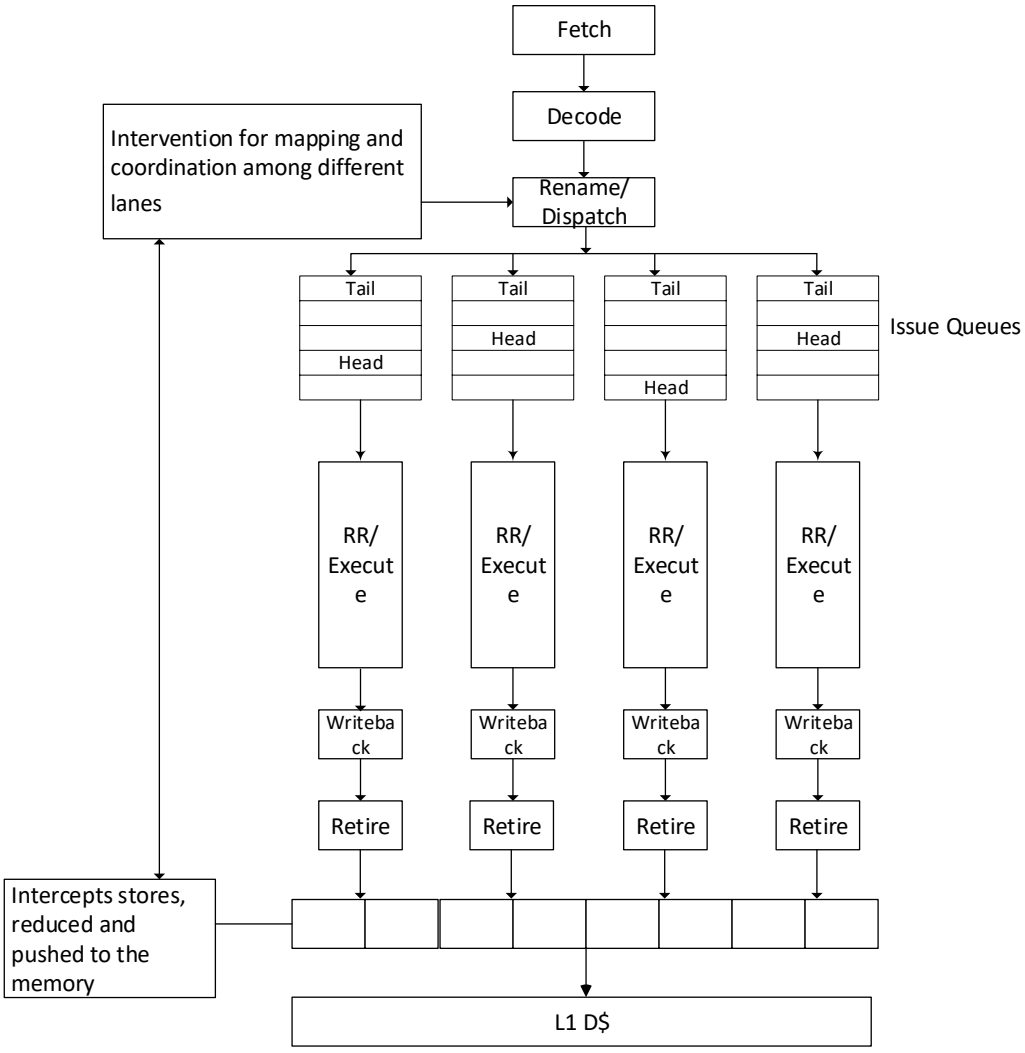

Figure 4: High-Level Vector Architecture with PSM interface

The processor fetches instruction sequentially unless backend stages redirects it to a different PC. Branch direction is solely determined by execution. There are multiple independent execution lanes each emulating one hardware thread. The decoded instructions are replicated and added to all the IQs which buffers these instructions for issuing.

Each lane is associated with an insn_pending_skip counter which helps avoid front-end squash in case of short forward branches. There is a per-lane stall signal that if asserted prevents IQ to issue any further instructions. A per-lane mask can disable any activity on that lane.

5. Overall Operation

We now discuss the key structures and overall working of the proposed architecture. We start in the scalar mode where only one lane operates. The PSM looks for certain PCs to transition into the vector mode. Once the instruction for that PC is issued, PSM introduces a barrier to ensure that the pipeline drains and the instruction commits. At this stage, PSM helps with mapping individual iterations to different lanes. Since, the lanes operate independently, they can reach the end of an iteration at different times. This can be due to both memory divergence and/or branch divergence. PSM monitors each thread to issue the last instruction in the loop and then asserts the per-lane stall signal which effectively acts as a barrier. PSM also intercepts all the stores from the lanes during the vector mode and pushes it to a local store list. Once all the lanes reach end of the loop, intercepted store list is reduced, and the values are pushed to the correct location. At this point the PSM checks if a next vector iteration or batch is required. If so, the live-ins (address in the input work list) are populated. Otherwise, PSM indicates a switch back to the scalar mode. Figure 5 depicts this high-level execution flow.

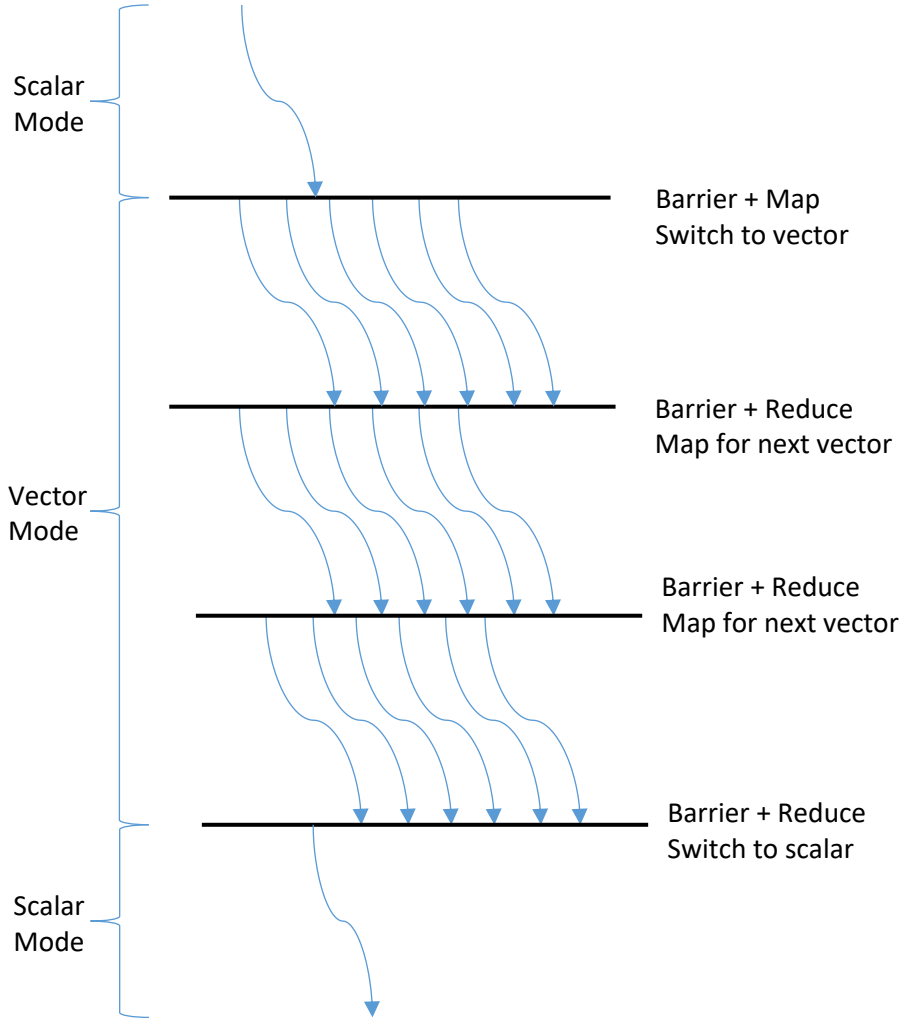

Figure 5: High-Level Execution Flow

5.1 Fetch/Decode Operation

Fetch and decode operation work similar to the OOO superscalar processor. The only difference is the absence of branch predictors. The fetch stage fetches instructions sequentially unless redirected to a different PC by backend stages.

5.2 Rename + Dispatch Operation

Since superscalar renaming structures (RMT, free-list) aren't required, rename and dispatch are combined into a single stage –rename_dispatch. Rename_dispatch replicates the

same instruction bundle at the tail of all the IQs. There is no real renaming, since each lane writes to its own set of register file and operates independently of other lanes. However, if one considers a centralized register file, duplication of instructions across the lanes would require just adding an offset per lane.

5.3 Issue-Queue Operation

Each cycle, at most one instruction issues from the head of each IQ. If issued, it reads register file, executes and updates the corresponding destination register corresponding to its own lane. Feedback paths and overall operation per lane is similar to that of an in-order pipeline. However, there are some key points to note:

- Tail pointer for all IQs are always in sync, and front-end stalls if even a single lane doesn't have sufficient space for the decoded instruction bundle.
- The lanes don't operate in lockstep. Each lane works independently, i.e. it makes its own decision whether to issue an instruction for a particular cycle or not. This is made possible due to replication of IQ. One strategy of vectorizing could be to have a single issue-queue with multiple head/tail pointers each corresponding to one execution unit. This would add multiple read ports to a single IQ. Another option could be to have a single IQ and have all the lanes operate in lockstep. Branch divergence can be handled in such an implementation using a SIMT-stack based approach or a predication-based approach (similar to GPUs). However, in both these approaches to handle divergence, each branch divergence adds up. Similarly, memory divergence also adds up i.e. all the lanes would stall if even one of the lanes misses in the cache. In the current implementation, each lane has its own program flow within a small window. Each lane executes its own loads, and therefore a cache miss on one lane doesn't stall another lane. We exploit MLP across lanes/iterations. However, since each lane executes in-order, the proposed architecture cannot exploit MLP within an iteration. Cache misses within one iteration are serialized.
- Branch direction doesn't depend on any prediction but relies on actual execution. When a branch instruction is issued from an IQ, no instructions are issued from the respective lane till the branch gets resolved. However, a lane never stalls because of a

branch in a different lane. By avoiding prediction of difficult branches, we avoid the problem of poor branch prediction. However, since multiple iterations of the same loop execute parallely in different lanes, we emulate a large speculative window.

- Branches resolve in the execute stage. If the branch destination is the sequential next PC, IQ is allowed to issue instruction at the head. If the next PC corresponds to a forward branch, IQ is checked if the decoded instruction is already present. Checking this doesn't require associative structure and is trivial. Difference between head and tail pointer indicate valid instructions in the IQ which also corresponds to sequential instructions in the binary. Difference in branch PC and branch destination indicates the number of instructions one needs to skip to reach the branch destination. We skip an equivalent number of instructions in the IQ, if valid instructions in IQ exceeds the branch destination offset. This requires incrementing the head of the respective IQ. This is shown in Figure 6. If the offset exceeds the difference between head and tail pointers, the head is reassigned to tail (indicating an empty IQ) and the difference is recorded in insn_pending_skip counter for that lane.

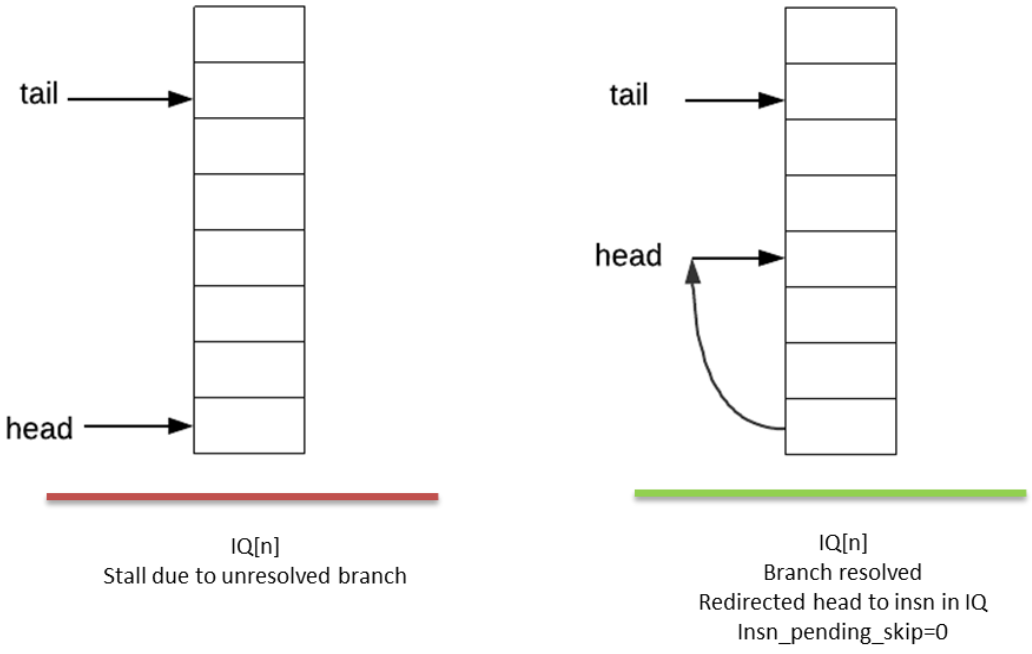

Figure 6: IQ Operation- Branch Destination in IQ

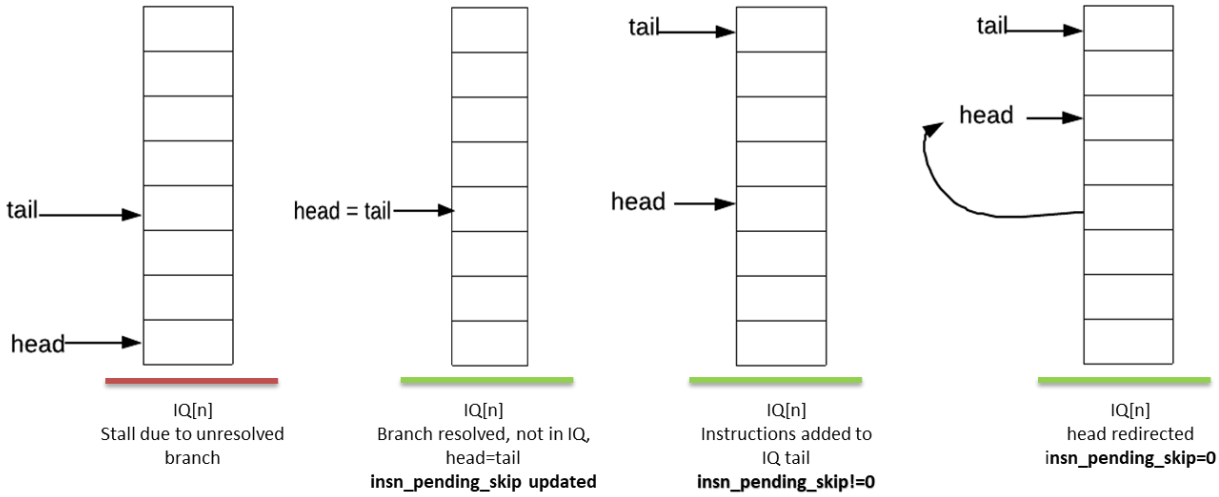

Figure 7: IQ Operation- Branch Destination not in IQ

- If instruction_pending_skip counter is not 0, respective IQ lane is checked to see the availability of the instruction. If available, head is reassigned correctly. If not available, head is reassigned to the tail and instruction_pending_skip is updated. This is shown in Figure 7.
- Due to the structure of the issue-queue, back branches are not supported in the vector mode. In the scalar mode, the IQ is completely cleared, and the fetch stage is redirected to the correct PC if a back branch is detected during execute.
- Load instructions can have variable latency due to hit/miss in cache. A load dependent instruction following the load can end up reading stale value from the register if the load suffered a cache miss. To avoid this, we take an approach similar to that taken for branch prediction. As soon as a load is issued, no further instruction from the same lane gets issued unless the load resolves. Note that this is different from the per-lane IQ stall.

5.4 Execute/Writeback/Retire Operation

The execute,writeback and retire stages retain most of their functionality. There is one key change in the execute stage: for branch operations, execute stage is capable of redirecting the

issue-queue head to point to branch destination. For back branches, it can redirect the fetch stage to a different PC.

Coupled with the above description of pipeline stages, the overall solution works in conjunction with PSM which helps with map and reduce phases.

5.5 Map Operation

When the start of a loop is detected, the processor transitions into the vector mode. There are multiple in-order scalar lanes. Each lane is responsible for executing one iteration of the loop. PSM is responsible for enabling the transition into the vector mode which involves correctly initializing the live-in registers for all the lanes. While most of the initialization involves just copying the architectural register file before the start of loop, PSM needs to initialize the iteration variable based on the iteration count. This helps vector lanes to work on their own elements parallelly.

In this work, we analyzed astar for such a transformation. Figure 8 and Figure 9 contain the disassembly of the astar object file compiled for RISC-V. To transform iterations over different lanes, PSM needs to track certain PCs which are listed below:

- pc_loop_addr_info: The loop iteration variable ‘i’ and updates to it are transformed into updates to the address in the input-work list upon compilation. This address corresponds to register x11(line 13 in Figure 8). In every iteration this address is incremented and checked against the last address in the input worklist. We operate a state-machine in the PSM which keeps track of this. We need to extract input worklist bounds for individual iterations to work correctly when mapped to different lanes. We extract this information using value of source and destination registers of pc_loop_addr_info. For the binary used, this corresponds to the PC of 0x17a30.
- pc_before_loop_start: As mentioned before, prior to loop start, the processor operates in scalar mode. To detect this, we need the PC just before loop start (0x17a34 for the astar binary we considered). PSM is expected to insert a barrier as soon as this instruction is issued. No further instructions are issued till this instruction retires. At this point, the map phase kicks off. During this phase, the architecture values are copied to register files for all the lanes. Since we also need to populate the ‘live-in’ value per lane (which is addres

in input worklist), PSM overrides value of register x11 based on base address extracted previously and the lane number. This is a based on trivial calculation:

x11(lane n) = ip_worklist base address + n*size_per_element.

A lane-mask is also calculated per lane which is responsible for disabling a lane. A lane is masked if its live-in input worklist address exceeds the upper bound calculated previously.

- pc_last_ins_loop: This corresponds to the last instruction in the loop (0x17d04, line 192 in Figure 9). Similar to pc_before_loop_start, once this instruction is issued, PSM inserts a barrier for the lane to avoid issuing any further instructions. This is required because we want to remove duplicate elements in the output worklist and need to wait till all the lanes are done with their respective work for that iteration. Once all lanes reach this point, we kick off the reduce phase to remove duplicate elements and push stores to the memory. There is another use of this PC. When in vector mode, once we add this instruction to the IQ, we redirect fetch to the first instruction in the loop. This is not required from a functional perspective and can be done during reduce phase also. It is assumed that the length of input worklist will be much more than the number of vector lanes. Therefore, multiple vector iterations will be required to complete iterating over the input worklist.
- pc_first_ins_loop: This corresponds to the first instruction in the loop (0x17a38) in our case. This is required to fetch the instructions for the next vector iteration. We do not reuse the instructions fetched for one batch of vector execution towards the next batch. A single iteration of the loop can be quite huge. To reuse the instructions, IQ needs to be atleast as large as the loop length which may not be physically viable.

As mentioned previously, PSM inserts a barrier for the respective lane when pc_last_ins_loop is issued. Each cycle, lanes are monitored if they have retired this instruction. Once all the lanes reach this point, the following two operations need to happen:

- Reduction of output worklist and propagating intercepted stores to the memory. Output worklist length is also updated (register x10) per lane. This is discussed more in the next section.

- Decision to run another vector iteration or transition back to scalar mode. The next address to be accessed in the input worklist is calculated per lane. If the addresses for the first lane exceeds the loop bound, we transition back to scalar mode. Otherwise, register x11 per lane (live-in for next iteration) is updated for the next batch.

```
0000000000017a0c <_ZN6wayobj10makebound2EPiiS0_>:       // start of the function
   17a0c: 00050793           mv  x15,x10
   17a10: 02052383           lw  x7,32(x10)
   17a14: 08053283           ld  x5,128(x10)
   17a18: 30c05063           blez  x12,17d18 <_ZN6wayobj10makebound2EPiiS0_+0x30c>
   17a1c: fff60e1b           addiw x28,x12,-1
   17a20: 020e1e13           slli  x28,x28,0x20
   17a24: 020e5e13           srli  x28,x28,0x20
   17a28: 001e0e13           addi  x28,x28,1
   17a2c: 002e1e13           slli  x28,x28,0x2
   17a30: 01c58e33           add x28,x11,x28
   17a34: 00000513           li  x10,0
   17a38: 0005a703           lw  x14,0(x11)            // start of the loop, x11 carries address of elem in input worklist(&B[i])
   17a3c: 0aa7d883           lhu x17,170(x15)
   17a40: 4077083b           subw  x16,x14,x7
   17a44: fff80f1b           addiw x30,x16,-1
   17a48: 000f0313           mv  x6,x30
   17a4c: 00231613           slli  x12,x6,0x2
   17a50: 00c28fb3           add x31,x5,x12
   17a54: 000fde83           lhu x29,0(x31)            // Indirect load access of form A[B[i]]
   17a58: 051e8263           beq x29,x17,17a9c <_ZN6wayobj10makebound2EPiiS0_+0x90>  // Load Dependent branch 1
   17a5c: 0787be83           ld  x29,120(x15)
   17a60: 00131313           slli  x6,x6,0x1
   17a64: 006e8333           add x6,x29,x6
   17a68: 00031303           lh  x6,0(x6)
   17a6c: 02031863           bnez  x6,17a9c <_ZN6wayobj10makebound2EPiiS0_+0x90>     // Load Dependent branch 2
   17a70: 00251893           slli  x17,x10,0x2
   17a74: 011688b3           add x17,x13,x17
   17a78: 01e8a023           sw  x30,0(x17)            // update output worklist
   17a7c: 0aa7d883           lhu x17,170(x15)
   17a80: 0015051b           addiw x10,x10,1           // update to output worklist length
   17a84: 011f9023           sh  x17,0(x31)            // Update to memory. Impacting later program flow
   17a88: 0a87d883           lhu x17,168(x15)
   17a8c: 011f9123           sh  x17,2(x31)
   17a90: 0a47a883           lw  x17,164(x15)
   17a94: 27e88c63           beq x17,x30,17d0c <_ZN6wayobj10makebound2EPiiS0_+0x300>   // Exit if reached destination
   17a98: 0aa7d883           lhu x17,170(x15)
   17a9c: 00281313           slli  x6,x16,0x2                                          // Computation for the next neighbour
```

Figure 8: wayobj::makebound2() RISC-V assembly excerpt 1

Figure 9: wayobj::makebound2() RISC-V assembly excerpt 2

5.6 Reduce Operation

While each loop iteration executes normally and updates its own register file, we do not allow it to update the memory state. The retire stage communicates store data/address to the PSM which buffers all the stores from all the lanes. The number and nature of these stores is specific to the application and PSM needs to be cognizant of that. PSM also maintains the order of all the stores. Since stores from one lane are in-order and the retire stage can communicate lane number when it pushes its store to the PSM fabric, keeping order of stores across the lanes is trivial. When all the vector lanes reach the end of the iteration, a PSM inserted barrier prevents them from making further progress. At this stage, the PSM fabric kicks off the reduction phase. While it is true that the reduction can start as early as the first 2 stores have been pushed to the store-buffer, we chose to start it only once all the lanes indicate that they have no work left for that vector iteration. In the context of astar, this reduce phase implies removing duplicate elements while maintaining the order of stores to the memory. We propose a fifo-based reduction scheme.

5.7 Fifo-based reduction

Consider store-buffer as a fifo similar to the one shown in Figure 10. At the start of each cycle, we look at the head of the store-buffer which indicates a valid element to store to memory. In each cycle, we push this store to the memory and we also broadcast the store to all the other entries in the fifo. Each entry compares the value of the broadcasted store with its own value and marks itself as invalid if there is a match, thereby removing duplicates. We move the head to the next valid element for the next cycle till the stage till there are no valid elements. Such a method requires as many cycles as unique elements are present in the buffer.

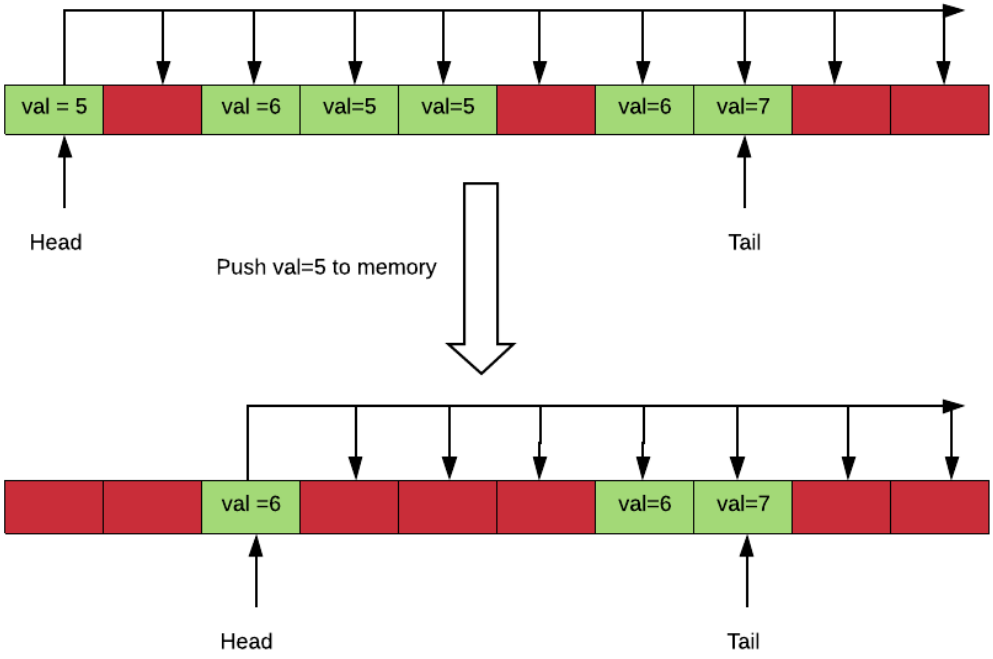

Figure 10: Fifo-based Reduction

6. Evaluation Environment

6.1 Baseline OOO Architecture

A canonical superscalar processor contains the following pipeline stages: Fetch, Decode, Rename, Dispatch, Schedule, Register Read, Execute, Writeback, Retire. The fetch stage contains logic for branch prediction and fetching instructions from the next predicted PC. Renaming stage renames instruction source and destination operands to correctly create data dependence flow among registers. Dispatch is the last in-order stage in the front-end and is responsible for adding instructions to the IQ and reorder buffer. The scheduler is allowed to issue any instruction to the backend, as long as its source operands are guaranteed to be available by the time instruction reaches the execute stage. Since the readiness of the instructions is unknown at compile time, the scheduler looks at all the instructions in the IQ to determine which instructions can be issued. Instructions executing are responsible for waking up dependent instructions in the IQ essentially establishing data-flow. Reorder buffer helps re-establish program order.

We used 721sim with the configuration listed in Table 1 as a baseline configuration for all the comparisons unless stated otherwise. 721sim is a detailed cycle-level execute-at-execute execution driven C++ superscalar processor simulator used in Prof. Rotenberg's research lab and in ECE721 Advanced Microarchitecture course. Big-window configurations refer to an active list of 1024, lsq of 512 and IQ size of 256. 721sim integrates both functional and performance simulator and is capable of running RISC-V instruction streams. It models the superscalar stages mentioned above. Functionally correct behavior is ensured by comparing retired instruction output against the functional simulator output.

Table 1: Baseline OOO Configuration

Baseline Superscalar Configuration			
Parameter	Size	Parameter	Size
activelist_size	512	l1_ic_line_size	64B
bp_table_size	65536	l1_ic_size	64KB
btb_size	4096	l2_c_assoc	8
checkpoints	64	l2_c_line_size	64B
dispatch_width	16	l2_c_size	256KB
fetch_queue_size	64	l1_ic_mshr	32
fetch_width	16	l1_dc_mshr	32
issue_queue_size	128	l2_c_mshr	32
issue_width	16	prf_size	576
l1_dc_assoc	4	retire_width	16
l1_dc_line_size	64B	lsq_size	256
l1_dc_size	64KB		
l1_ic_assoc	8		

6.2 Vector + PSM Configuration

721sim formed the base for the wide-vector micro-architecture and map-reduce modeling. PSM functionality was emulated by tracking key PCs in different pipeline stages and modeling expected behavior. This involves functions such as mapping of vector iterations, intercepting stores, reducing stores and adding barriers. The store-buffer was configured to match requirements of astar application. Table 2 lists the baseline configuration used for the wide-vector architecture. While PSM can be used towards targeting certain sections of the application, in this work, we use the vector processor even outside of the wayobj::makebound2() function. To ease our implementation, two modes were implemented: vector and scalar. When in

scalar mode, only lane 0 is active. In vector mode for baseline configuration, lane 1 to lane 16 operate parallelly. Therefore, there are a total of 17 lanes which is reflected in the Physical register file size. However, we use a maximum of 16 at a time. We have chosen 16 lanes for baseline vector configuration to match with the issue-width of 16 in the superscalar processor.

Table 2: Vector Processor Configuration

Baseline Vector Configuration			
Parameter	Size	Parameter	Size
Number of Lanes	16	l1_ic_line_size	64B
Store-buffer size (PSM)	384	l1_ic_size	64KB
dispatch_width	16	l2_c_assoc	8
fetch_queue_size	64	l2_c_line_size	64B
fetch_width	16	l2_c_size	256KB
issue_queue_size (per lane)	64	l1_ic_mshr	32
issue-width (per lane)	1	l1_dc_mshr	32
l1_dc_assoc	4	l2_c_mshr	32
l1_dc_line_size	64B	prf_size	1088
l1_dc_size	64KB	retire_width (perlane)	1
l1_ic_assoc	8	Fifo-based reduction	

6.3 Functional Correctness and Performance Measurement

While in vector mode, we ignore the store-load dependencies and may execute sections of code that weren't supposed to be executed. PSM helps us to recover from this. However, the functional model is left unchanged. Therefore, it is non-trivial to compare the output of every committed instruction against the functional simulator output. In the current implementation, we do two checks to ensure functional correctness:

- Checking each instruction on retire is turned off in the vector mode. At the start of vector mode, the functional simulator is indicated to run ahead till the end of loop exit. All the results from the functional simulator that we typically compare are ignored. The performance simulator runs without any checking while in the vector mode. When the performance simulator switches back to the scalar mode, the functional simulator is in sync and every instruction executed in scalar mode is compared for functional correctness at this point. One of the key outputs of the loop is the output worklist length. Any functional incorrectness impacting output worklist gets flagged immediately.
- For the purpose of this work, updates to output worklist were extracted from the baseline OOO configuration by dumping the address, size and value to a file. In the vector implementation, reduction phase pushes stores to the memory. We dump similar information to a file and compare it to ensure that the output worklist was updated correctly.

Another area where this implementation differs from the baseline OOO implementation is keeping track of dynamic instructions executed. This is because of the recovery scheme employed. In the baseline OOO machine, a branch misprediction is handled via squashing instructions following the mispredicted branch. Speculatively executed instructions get counted only if they retire or in other words the assumption under which they executed holds true. For the vector implementation, knowing the loop behavior allows us to violate store-load dependencies and use PSM to recover from it. Therefore, we might end up throwing some extra work that wasn't required, but we don't squash instructions. We don't track the control flow that should have resulted if we didn't violate these dependencies. Therefore, to find instructions that should have executed, we augment both the superscalar simulator and vector simulator to count number of times function wayobj::makebound2() executes. This was done by analyzing the binary and finding PC corresponding to function exit. We extract the instruction count from the superscalar processor for the same function exit count to find true IPC in the vector machine. All the results correspond to 10,000 calls to wayobj::makebound2() function which corresponds to nearly 100 million dynamic instructions.

7. Results and Analysis

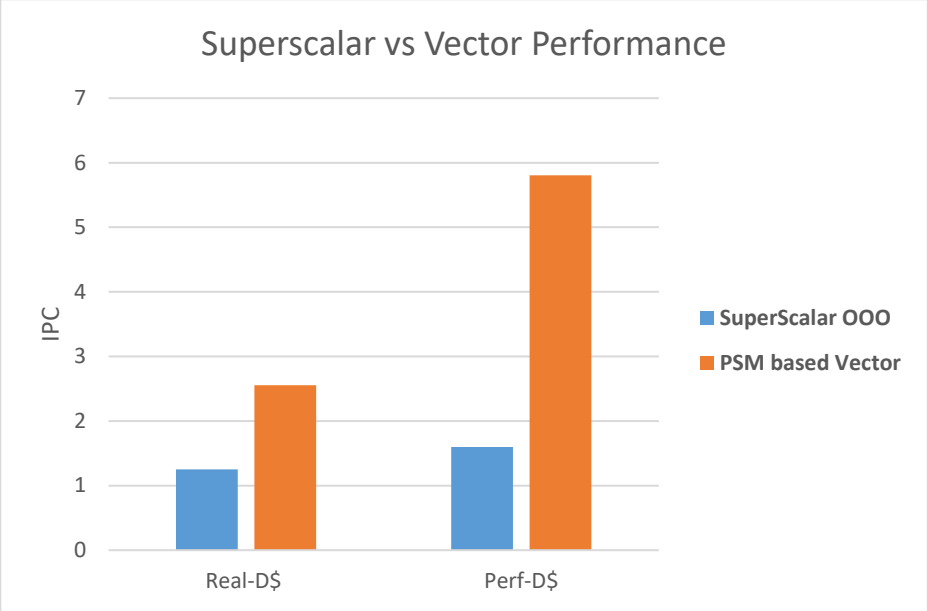

Figure 11: IPC comparison for baseline configurations

Figure 11 compares the IPC for baseline configurations with real and perfect L1-D Caches. For real L1 D-Cache configuration we see that the vector architecture with fifo-based reduction yields a considerable IPC boost (2.04x) over the baseline superscalar OOO processor. With perfect L1 D-Cache, the IPC reaches 5.8 in the vector architecture as compared to 1.6 in the superscalar processor. The modest increase in IPC with perfect L1 D-Cache in superscalar can be attributed to the high branch misprediction rate. With a perfect L1 D-Cache, branches resolve faster and therefore the penalty one pays for branch misprediction is less. However, one still squashes every instruction behind the mispredicted branch and refetches the instruction from the correct path. Figure 12 presents a clear picture of this behavior. We see that biggest boost in performance in baseline superscalar is from perfect branch predictor configuration. A baseline configuration with a bigger window doesn't yield any benefit for the same reason. A bigger instruction window helps if the work done speculatively in the shadow of the long latency operation is correct. However, in case of astar, the speculative work gets squashed due to mispredicted branches. We also see that the vector baseline configuration performs better as compared to both the perfect L1 D-Cache and big-window superscalar configurations.

Superscalar configuration with perfect L1 D-Cache and perfect branch prediction shows a 2% gain over the perfect L1 D-Cache configuration of the vector configuration. While the superscalar is expected to outperform the vector processor in this configuration, it doesn't yield as much benefit because of the fetch-breaks. The vector processor does extra work, but with the multiple issue queues essentially buffering multiple iterations of the loop at the same time, it doesn't suffer from breaks in the fetch bundle. If one includes perfect fetch in the superscalar processor, we observe the IPC jump to 15.98.

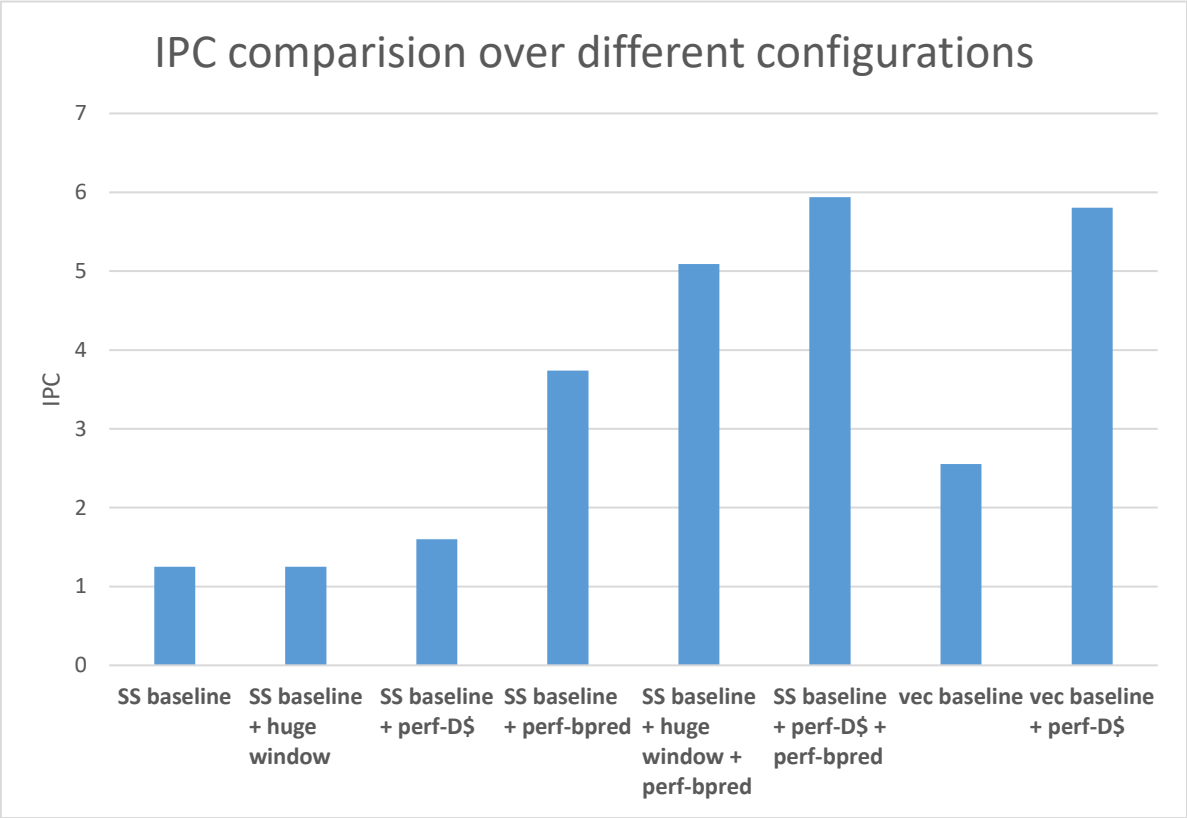

Figure 12: Performance trend over different superscalar and vector configurations

If we consider perfect L1 D-Cache configuration in the vector processor, we see a significant jump in the IPC. Comparing against the baseline vector configuration, the jump in performance is because of the barrier inserted at the end of each vector iteration. In the baseline vector configuration, the slowest lane stalls all the other lanes. Cache misses within a lane are serialized since individual lanes operate in-order. Therefore, a lane that suffers multiple cache misses prevents other lanes from making forward progress. Since the IQ size doesn't capture the

complete loop iteration, a faster lane could also starve for instructions since IQs are synchronized at the tail. Perfect L1 D-Cache helps avoid this behavior. Imbalance across the lanes could only result from branch divergence when perfect L1 D-Cache is assumed.

We also analyze how reduction cycles impact IPC. In Figure 13, we assume abstract reduction cycles of 0, 10, 50 and 100 per vector iteration for the baseline configuration with 16 vector lanes. As expected, IPC drops as reduction cycles are increased. However, even with 100 reduction cycles per vector iteration, the IPC drops by 18% as compared to 0 reduction cycles per iteration.

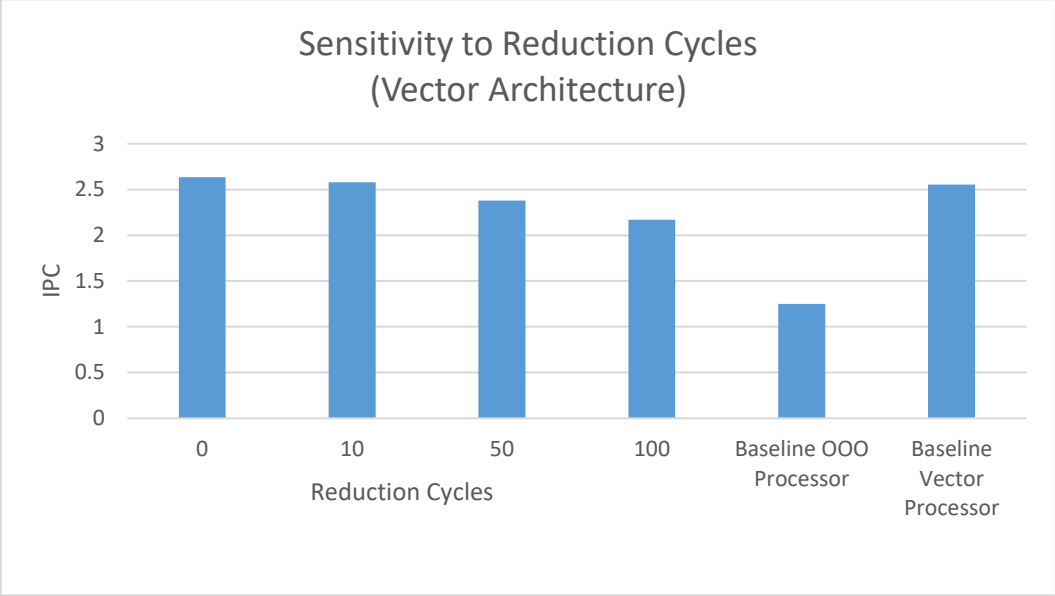

Figure 13: Sensitivity Analysis: Reduction Cycles

Figure 14 shows the average unique and duplicate elements per iteration for different vector lane configurations. This is independent of other parameters and reduction algorithm employed. For the configuration with 16 lanes, we see average 12.66 duplicate elements and average 15 unique elements which points to an average unreduced list size of ~28 per vector iteration. If one considers a hypothetical reduction algorithm, that takes an average of 100 reduction cycles, the vector architecture still provides an IPC improvement of 72%.

Figure 14: Average unique and duplicate elements per vector iteration

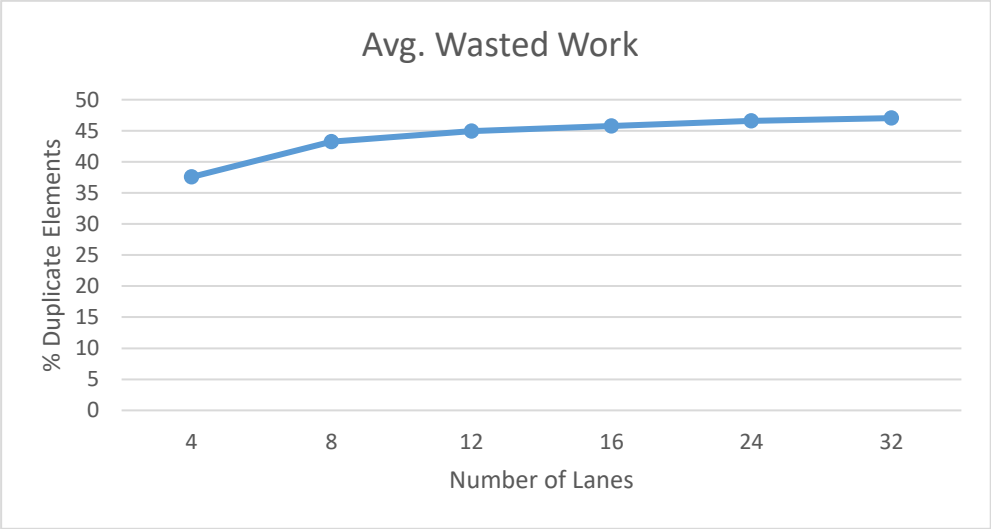

Figure 15: Average wasted work per vector iteration

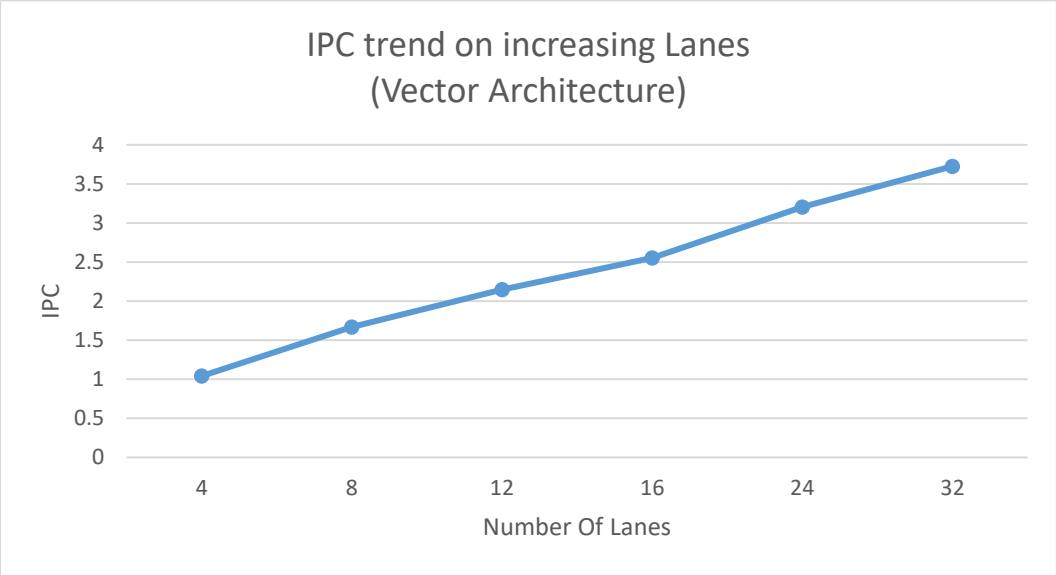

Figure 16: Impact of increasing number of lanes

We also analyze the impact of increasing lanes. In Figure 15, we see that around 37% of the elements are duplicate with 4 vector lanes, while for 32 lanes, the duplicate elements reach to around 47%. We see that the percentage of duplicate elements doesn't increase as much on increasing the lanes. The duplicate elements in the output worklist are indicative of the duplicate work due to violation of store-load dependencies across iterations. Since the percentage duplicate work doesn't increase as much, increasing number of lanes would work well as the speculative window does considerable useful work. Note however, that this is because both useful work and wasted work increase in a similar fashion (Figure 14). In Figure 16, we see that the IPC increases from 1.04 for 4 lanes to 3.72 for 32 lanes.

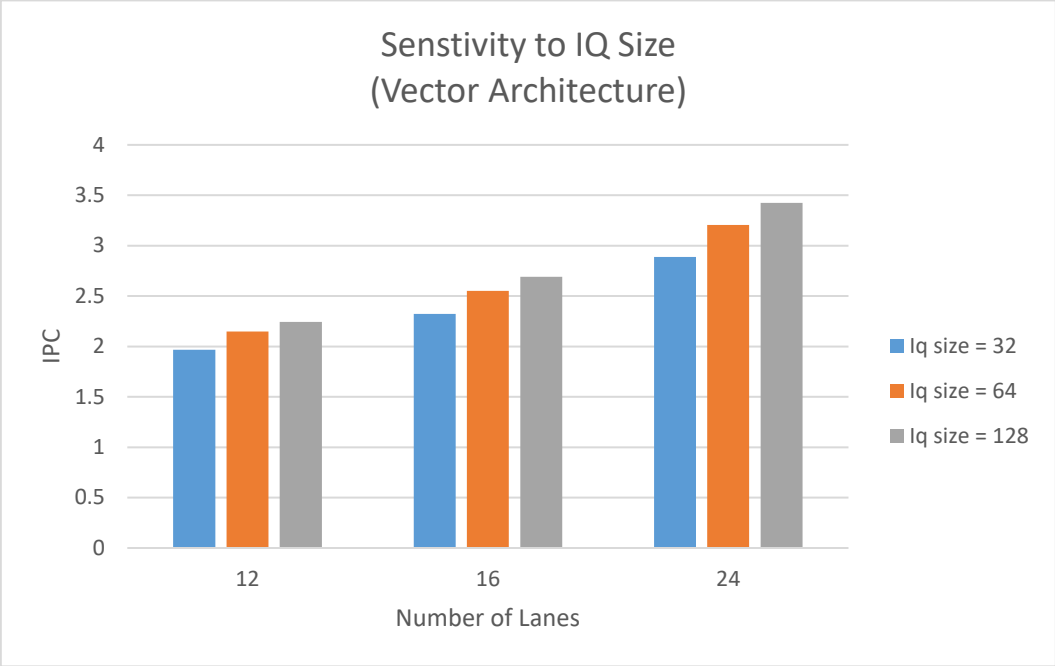

Figure 17: Impact of IQ-size

IQ is a key structure for both superscalar and the vector architecture. For superscalar, it provides a window to find independent instructions. Increasing IQ size helps with superscalar architecture if the application lacks ILP in small instruction window and needs a larger window to find far flung ILP. However, for a benchmark like astar, increasing size of IQ doesn't help because of branch mispredictions. If one includes perfect branch-prediction, work done in shadow of unresolved branch doesn't get squashed and a bigger window helps.

For vector architecture, multiple IQs help with independently running iterations. Increasing IQ size helps avoid stalls in execution. A lane with forward branches to larger offset would get stalled if the IQ is not capable of handling the offset. Since the tail of the IQs are always in sync, the lane with branch stalls till all the other lanes make significant forward progress so that more instructions could be added. Similarly, even a single slow lane (due to cache misses) stalls all the other lanes towards making forward progress. In Figure 17, we see that the IPC increases by upto 18% as the IQ size is increased from 32 to 128.

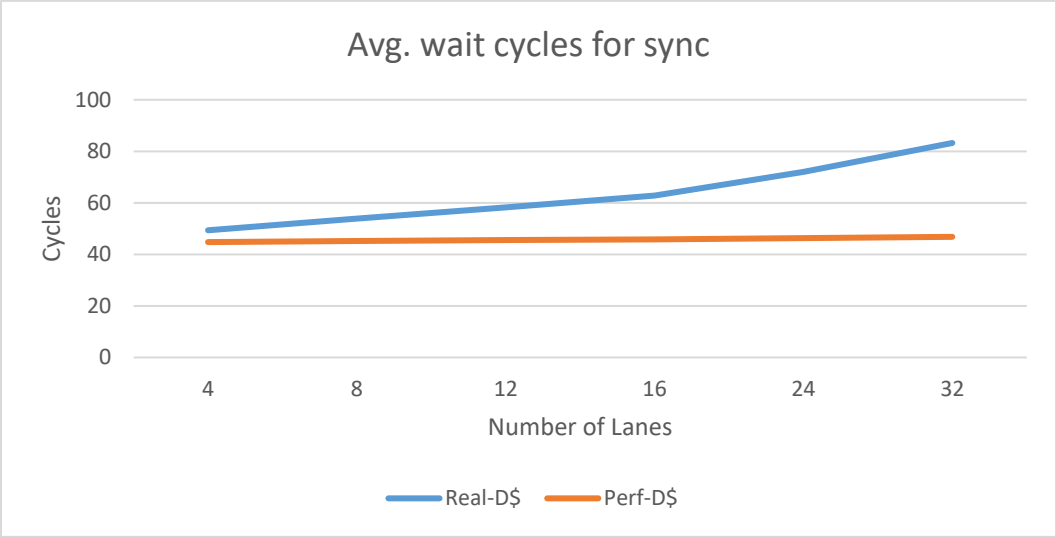

Figure 18: Average wait cycles at barrier

There are two sources of divergence among the different lanes that operate parallelly: branch divergence and memory divergence. This leads to an imbalance across the lanes where the fastest lane waits for the slowest lane. Since the lanes are allowed to operate independently within the limited window, branch divergence translates into different number of instructions being executed by different lanes. In Figure 18, we see the impact of this divergence. We measure the average number of cycles the fastest thread waits for the slowest thread i.e time spent at the barrier. We see that for the perfect L1 D-Cache configuration, cycles waited by the fastest thread doesn't increase much. With a perfect L1 D-Cache, the only source of divergence is branch divergence and its impact doesn't increase much with increase in number of parallely operating threads. For the real L1 D-Cache configuration, we see that the cycles spend waiting increase with the number of lanes. This corresponds to memory divergence. As we increase the number of lanes, the chance of multiple cache misses in a single lane increases. With the proposed vector architecture, while we parallelize cache misses across the lanes, cache misses within a lane are serialized.

8. Conclusions and Future Work

8.1 Conclusions

The intent of this work was to present a potential use-case for Post silicon micro-architecture. We specifically chose wayobj::makebound2() function of SPEC 2006 benchmark astar for our work. We did an analysis of the same function to identify key PCs. This helped us identify key points of intervention for the PSM. We implemented a wide-vector architecture capable of running independently running hardware threads. This included a different structure of the issue-queues to enable a huge speculative window and help avoid memory and branch divergence. We were able to map the iterations of the astar loop to different lanes. We also modeled and discussed a hardware-based reduction algorithm.

The work shows a different dimension of extracting IPC from applications. We demonstrated an architecture that leverages PSM to overcome the problem of poor branch prediction. It is able to parallelize loop iterations despite the presence of loop-carried dependencies such as those that plague wayobj::makebound2() function of astar.

We were able to show IPC boost of ~2x for a vector configuration with real caches and about 3.625x with perfect L1 D-Cache.

8.2 Future Work

During the course of this work, we realized multiple areas of opportunity in which this work can be continued further. These are:

- Prefetcher : Since we wait for all the lanes to finish with one vector iteration before reduction and starting the next one, the slowest lane impedes other lanes. In the previous section, we saw that the vector architecture leaves a lot of IPC potential on the table due to cache misses. Therefore, implementing prefetcher is a natural direction to improve the performance. Among various options there are 2 directions that we are interested to explore:

    - Indirect Memory Prefetcher: As discussed before, there are indirect loads in astar of the form A[B[i]] which do not exhibit a regular pattern and therefore are difficult to predict. IMP[8] seems ideal for this situation and we would like to explore how it performs with the vector architecture for astar.
    - Execution based prefetcher: Another interesting direction that we want to explore is the execution based prefetcher. We take certain cycles for reduction during which the vector lanes remain idle. The idea here is we can initialize the live-ins for the during the reduction phase and let the execution proceed. We can provide fake values on load-misses and avoid committing stores to the memory. Since address computation for each of the neighbor is independent of the previous neighbor computation, we can get an accurate execution based prefetcher.
- Hardware implementation of the reduction hardware: The current fifo-based reduction hardware was modeled in C++. We would like to model the same in Verilog and do a feasibility study. Additionally, we want to explore other reduction hardware algorithms with different area/performance cost.
- Automating map phase: For our work, we manually analyzed the benchmark and extracted key PC's. This will be tedious if done manually for each application. We would like to come up with a framework that helps in identifying these key PCs. If possible, we would also like to automate this process.
- Store-load dependencies within loop iterations: The stores for each iteration are intercepted by PSM and buffered. While we willfully ignore loop-carried dependencies and fix them via reduction, we also need to be careful about the store-load dependencies within an iteration. For astar, stores and loads within one iteration are bound to access different locations since they correspond to different iterations. This will not be the case for all the loops. Therefore, we would like to explore a feasible way to avoid violating store-load dependencies within a loop iteration.
- Morph-Core[2] implementation: In this work, while we used multiple execution lanes in the vector mode, only one in-order lane was used in the scalar mode. Within the PSM paradigm, we assume a heterogenous multi-core system. However, if one compares superscalar and the vector architecture, a lot of hardware remains same. Key changes involve IQ and renamer. We would like to explore a Morph-Core based architecture

where the same superscalar processor behaves like a wide-vector architecture. We would like to still leverage the idea of PSM to help with map and reduction phases. PSM could also possibly help with transforming superscalar to vector architecture behavior.

- Energy/Performance aware lane activation: The current implementation enables as many lanes as required for each vector iteration. However, it doesn't monitor the performance it gets out of enabling more lanes. It doesn't incorporate any energy measurements. One could dynamically control the maximum number of lanes that will be enabled based on such metrics. Amount of duplicate work done could be an important metric, which could be measured by redundant elements in the output worklist. Another important metric could be finding median time to completion per lane.
- Implementing superscalar based PSM assisted reduction forcing all branches to compute all the elements: While the amount of redundant work increases in such an implementation, the only bottlenecks here will be cache misses and reduction functionality. However, we could extract MLP since cache misses don't serialize. By devising an efficient reduction algorithm, we might be able to extract more performance.
- Exception handling: Handling exception in the discussed implementation is simple because of separation of scalar and vector modes. When in scalar mode, the processor can handle the exception normally since there is no speculative work. While in vector mode, we can update live-outs of the iteration at the end of each vector iteration that executes without any exceptions. During the execution of a vector iteration we do not modify lane 0 register file. Since lane 0 maintains architectural state at end the last vector iteration, if an exception is detected during the vector mode, we can revert to scalar mode with PC pointing to start of the loop. This would allow us to process the exception in scalar mode.
- Lockstep operation to exploit MLP: Load driven branch conditions in astar determine the control flow. However, with the help of PSM functionality, we violate store-load dependencies across iterations and as a result, we also violate control flow. Workload across the lanes is unbalanced because of memory and branch divergence. With the knowledge of PSM, we could force all the 8 computations to always happen (by overriding the load values). This would allow us to operate all lanes in lockstep and therefore use a single IQ. The results of the load and therefore the branch can be used to individually squash buffered stores.

9. References

[1] Muawya Al-Otoom, Elliott Forbes, and Eric Rotenberg. 2010. EXACT: explicit dynamic-branch prediction with active updates. In *Proceedings of the 7th ACM international conference on Computing frontiers* (CF '10). ACM, New York, NY, USA, 165-176. DOI: https://doi.org/10.1145/1787275.1787321

[2] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson and Y. N. Patt, "MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP," *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Vancouver, BC, 2012, pp. 305-316.

[3] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2016. A Survey on Thread-Level Speculation Techniques. *ACM Comput. Surv.* 49, 2, Article 22 (June 2016), 39 pages. DOI: https://doi.org/10.1145/2938369

[4] Lawrence Rauchwerger and David Padua. 1995. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* (PLDI '95). ACM, New York, NY, USA, 218-232. DOI=http://dx.doi.org/10.1145/207110.207148

[5] Ye Zhang, L. Rauchwerger and J. Torrellas, "Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors," *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*, Las Vegas, NV, USA, 1998, pp. 162-173.

[6] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," in *IEEE Transactions on Computers*, vol. 48, no. 9, pp. 866-880, Sept. 1999.

[7] J. T. Oplinger, D. L. Heine and M. S. Lam, "In search of speculative thread-level parallelism," *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*, Newport Beach, CA, USA, 1999, pp. 303-313.

[8] X. Yu, C. J. Hughes, N. Satish and S. Devadas, "IMP: Indirect memory prefetcher," *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Waikiki, HI, 2015, pp. 178-190.

[9] R. Sheikh, J. Tuck and E. Rotenberg, "Control-Flow Decoupling," *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Vancouver, BC, 2012, pp. 329-340.

[10] R. Sheikh, “Control-Flow Decoupling: An Approach for Timely, Non-speculative Branching. (Doctoral dissertation).” Retrieved from http://www.lib.ncsu.edu/resolver/1840.16/8736

[11] E. Rotenberg, “FoMR: Post silicon micro-architecture, NSF grant no. CCF-1823517”, 2018.

[12] Matthew A. Watkins and David H. Albonesi. 2010. ReMAP: A Reconfigurable Heterogeneous Multicore Architecture. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO '43). IEEE Computer Society, Washington, DC, USA, 497-508. DOI: https://doi.org/10.1109/MICRO.2010.15

10. Appendix

10.1 Early Returns

Considering wayobj::makebound2() loop, we see that there are 2 ways of exiting the loop and the function:

- Reaching end of input worklist
- Reaching destination: The function returns from the neighbor that reached the destination after marking the flend flag (Line 29 in Figure 1).

When transforming the loop into a map-reduce form, we need to be able to detect if during execution, destination node was reached. This is necessary for 2 reasons:

1. Stores beyond this point need to be ignored for reduction and updating the memory.
2. Irrespective of whether we have reached the end of input worklist or not, we need to transition back to scalar mode and exit the loop.

To handle this properly, any lane reaching the destination communicates this to the PSM fabric with its lane number. PSM fabric already maintains the global order of stores within its buffers. Therefore, during reduction, it knows the bound of elements which need to be considered for reduction. Other stores can be discarded.

10.2 Output Worklist Address

In vector mode, since multiple iterations of the loop run in parallel, a younger iteration intending to update the output worklist, doesn't know how many updates the older iterations will make. That is, consider just two lanes, where each could add upto eight entries in the output worklist. Since lane 2 runs in parallel with lane 1, it doesn't know where to make updates in the output worklist since this depends on the total updates that lane 1 makes. We leverage PSM to help us in this aspect. While PSM intercepts address and data of the stores across all the lanes, it can disregard the address provided by the lanes. The address from first store can be used for all the other addresses in the output-work list. In our implementation, PSM saves this address and increments this for every valid store to the output worklist post reduction.

10.3 Store Buffer Length

The store-buffer in PSM fabric needs to buffer all the stores for all the iterations that are running in parallel. For astar, a single iteration looks at 8 neighbors. If the neighbor was not visited before, there are three stores for this neighbor. In the worst case, there would be 24 stores per iteration. Assuming a 16 lane wide architecture, this would mean a store-buffer with 384 entries. However, the reduction doesn't happen over all these entries even in the worst case. This is because only one of the three entries correspond to the output worklist. These can be seen as conditional stores dependent on the update to the output worklist. We can separate these two types of stores into two buffers since the first store per neighbor undergoes reduction and we broadcast it to other elements in the list for invalidation of duplicates. If an element is marked as duplicate, the other two stores corresponding to the same neighbor aren't propagated to the memory.