ABSTRACT

MATHUR, UTKARSH. Post-Silicon Microarchitecture (PSM) Implementation of Checkpointed Early Load Retirement (CLEAR). (Under the direction of Dr. Eric Rotenberg).

Post-Silicon Microarchitecture (PSM) refers to instantiating new or uncommercialized microarchitectures on top of flagship superscalar and vector cores. Reconfigurable fabric like Field Programmable Gate Array (FPGA) or Coarse Grained Reconfigurable Array (CGRA) might be used as a PSM substrate which instantiates these microarchitectures. PSM is all about microarchitecture interventions and observations for the reconfigurable hardware to boost overall performance of the system.

We explore Checkpointed Early Load Retirement (CLEAR) as the underlying microarchitecture that the PSM implements and propose a design which we refer as PSM CLEAR. We also define a PSM interface that the processor needs to support to enable PSM CLEAR microarchitecture.

Detailed simulations of our proposed design across different applications from the SPEC2006 and SPEC2017 benchmark suite reveal a 7.05% and 35.12% geometric mean speedup with a realistic load value prediction machinery and a perfect load value prediction machinery respectively relative to a contemporary out-of-order processor.

Post-Silicon Microarchitecture (PSM) Implementation of Checkpointed Early Load Retirement (CLEAR)

by
Utkarsh Mathur

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. Huiyang Zhou	Dr. James Tuck

Dr. Eric Rotenberg
Chair of Advisory Committee

BIOGRAPHY

Utkarsh Mathur was born in Agra, India in 1994. He did his schooling in Agra and graduated high school from St. Clares Sr. Sec. School. He is a merit certificate holder in Science from Central Board of Secondary Education (CBSE), India for being in the top 0.1% of students who appeared in All India Senior School Certificate Examination (AISSCE 2009). In 2011, he joined Jaypee Institute of Information Technology (JIIT), Noida, India to pursue Bachelor of Technology (B. Tech) in Electronics and Communication Engineering. During his undergraduate studies, he also worked on several research projects in various fields like Micro-Electro-Mechanical Systems (MEMS), Computer Vision (CV) and Image Processing (IP). He graduated from JIIT, Noida in 2015 and joined the Verification IP (VIP) team at Cadence Design Systems in Noida, India as a Research & Development Engineer. During the period of 2 years at Cadence, Utkarsh worked on multiple projects like HDMI, MHL, USB type-C. However, he was still looking to delve deeper into computer architecture and decided to pursue graduate study in the USA. He was accepted to the Computer Engineering Master's program at North Carolina State University (NCSU) in fall 2017 after which he took the opportunity to work under the supervision of Dr. Eric Rotenberg towards his Master's thesis.

His primary research interest is in computer architecture with special interests in high-performance microarchitecture, General Purpose Computation on Graphics Processors (GPGPU), and architectural support for security. Upon completion of his graduate degree, Utkarsh plans to join Marvell Technology Group, Santa Clara as an Architecture Engineer.

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Dr. Eric Rotenberg. He has been a constant source of support and inspiration since the day I began my research. Thank you for your guidance and encouragement. Thank you for supporting my ideas and steering me in the right direction.

I would thank my committee members Dr. Huiyang Zhou and Dr. James Tuck for supporting me in this work. A special thanks to Dr. Huiyang Zhou for supporting me in the work beyond this thesis and motivating me to think in different research directions. I would also like to thank many faculty members in the ECE department who have helped me either in class or otherwise.

I would like to thank my friends, Aayush Chaudhary, Adith Vastrad, Mansi Jain and Saransh Jain who have always been more than willing to help.

Finally, I would like to thank my parents (Samir Gopal Mathur & Vidula Mathur) and my brother (Udit Mathur) for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching. This accomplishment would not have been possible without them.

This thesis was supported in part by NSF grant No. 1823517 (FoMR: Post Silicon Microarchitecture), 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

CHAPTER

1

INTRODUCTION

In the past, computer architects have been delivering significant improvements in the CPU performance by employing clever microarchitectural techniques and leveraging the advances in the process technology. With the breakdown of Dennard scaling, speed improvements from technology scaling have slowed down resulting in computer architects to explore improvements in both microarchitecture and architecture. Heterogeneous computing is gaining popularity as it offers performance improvements that a CPU alone might not be able to achieve.

Heterogeneous systems which includes GPUs, custom accelerators, etc., might require changes in the application binary to embed the information for parallelization. Reconfigurable fabrics that are connected to the CPUs, and configured as the application demands might be the way to meet application specific speedup targets.

1.1 Post-Silicon Microarchitecture

In this work, we use the paradigm of Post-Silicon Microarchitecture (PSM) [2] which refers to instantiating a new or uncommercialized microarchitectures on top of a general-purpose

instruction-level parallel (ILP) core. Reconfigurable hardware like Field-programmable Gate Array (FPGA) or Coarse Grained Reconfigurable Array (CGRA) which we refer to as PSM substrate in the text might be used as a building block to instantiate the new or uncommercialized microarchitecture.

PSM is fundamentally different from the hybrid Von Neumann / programmable hardware fabrics as it does not require code fragments to be compiled for one or the other. PSM is all about microarchitecture intervention and observation, ranging from dynamically instantiating new types of predictors and caches, new components to support different execution modes, or logic that changes the way instructions in the conventional processor flow and interact.

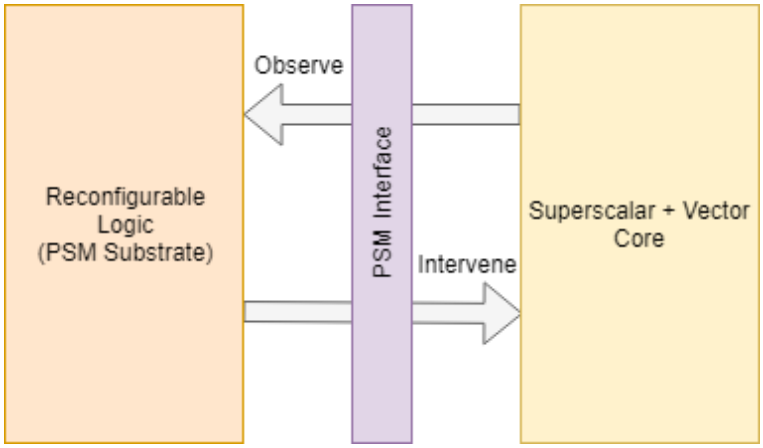

Figure 1.1: Post-Silicon Microarchitecture

Many microarchitectures that have shown to increase Instructions Per Cycle (IPC) significantly on various applications have never been implemented in commercial processors because of narrow workload applicability, cost and risk (e.g., design and verification effort) associated with them. Such microarchitectures are typically the best candidates for a PSM based implementation as they are not hardened onto the silicon and are configured on-the-fly. An application, which profits from a particular execution model, programs the PSM substrate to implement a custom component required by the execution model and programs the interface between the PSM substrate and the pipeline stages for necessary coordination.

1.2 PSM Interface

In order to orchestrate the interplay of data and control between the processor pipeline and the PSM substrate, it is required to define a signal interface or PSM Interface (PSMI) which acts as a channel for the processor and PSM substrate to interact. These interfaces define the degree of interventions in a processor pipeline, hence, sets the limit to microarchitectures that can be implemented on the PSM substrate. The design of these interfaces is critical as they become a part of the processor design and are hardened onto the silicon. These interfaces may or may not be exposed as a part of the ISA.

Any microarchitecture that is implemented on the PSM substrate might only use these interfaces to communicate with the processor. An example of such an interface could be “squash" which if asserted from the PSM substrate causes the processor to squash everything in its pipeline and start fetching from a program counter (PC) that the PSMI supplies.

1.3 Checkpointed Early Load Retirement (CLEAR)

Checkpointed Early Load Retirement (CLEAR) [3] is a complexity-effective technique to get an illusion of a large window processor by using load value prediction (LVP) [4] [5] [6] [7] [8] [9] [10] [11] and architectural state checkpointing. CLEAR mode is triggered when a long latency unresolved load reaches the head of active list or reorder buffer (ROB). The processor checkpoints the architectural registers, and supplies a load value prediction to the unresolved load. Dependent instructions are woken up followed by early/fake/pseudo retiring the value predicted load. This mechanism unclogs the active list and issue queue, giving the younger speculative instructions an opportunity to execute early.

When the value of load returns from the memory subsystem, it is compared against the predicted value. In the case of a load value misprediction, processor’s state is rolled back to the checkpointed state, discarding all the speculative computations. In the case of a correct prediction, the checkpoint is simply freed and the fake committed work is deemed to be correct.

In order to support CLEAR mode, the processor needs support of at least one architectural checkpoint. Multiple checkpoints might allow the processor to early-retire multiple unresolved loads, each of which may or may not be guarded by a checkpoint (coarse-grained checkpointing based on confidence).

In CLEAR, a predicted value is provided to the unresolved load when it becomes most critical, i.e., when it reaches the head of active list. The design choice of not providing a

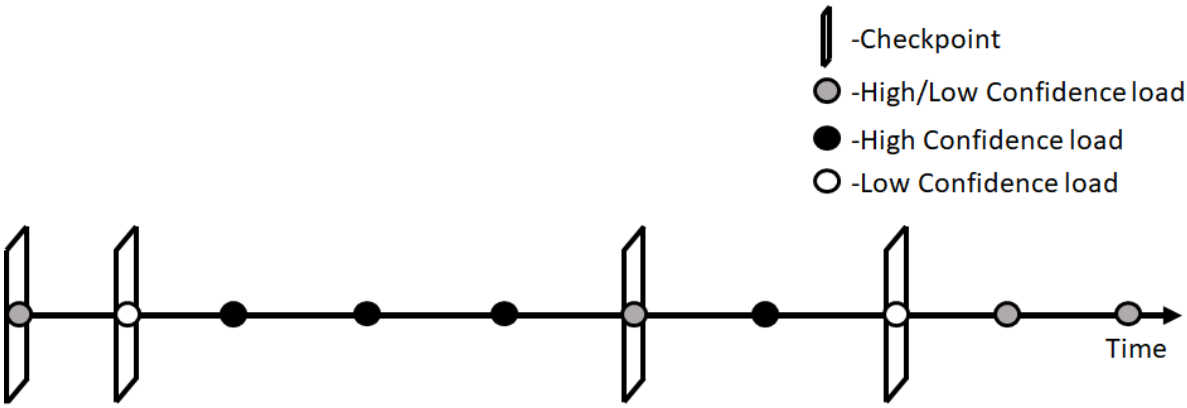

Figure 1.2: Coarse grained checkpointing in CLEAR based on confidence; CLEAR mode proceeds even if the system runs out of checkpoints

load value prediction early in the pipeline might limit the performance of CLEAR.

In this work, we re-design the CLEAR microarchitecture for a PSM based implementation taking into account the cost of intervention in the processor required to support it. We also design an interface that the processor supports to enable PSM CLEAR.

1.4 Related Work

ReMap [12] is a reconfigurable architecture which is coupled with a cluser of cores. The work requires applications to be explicitly parallelized. Portions of the communication and computation among threads are accelerated in the reconfigurable fabric. PSM differs from the ReMap work as we do not require explicit parallelization and modifications in the application source. Our reconfigurable fabric observes and intervenes within the microarchitecture to orchestrate customized parallel execution.

Authors in [13] propose Smart Memories, a modular reconfigurable architecture comprised of many processing tiles co-mingled with many mats of memory. PSM differs from Smart Memories as it uses fine-grain programmable hardware to intervene within and among the core's pipeline stages.

CLEAR [3] is a mechanism that provides the illusion of a large window processor by using load value prediction and architectural checkpointing. PSM CLEAR differs from the traditional CLEAR in two ways. First, CLEAR provides load value prediction at retirement, missing the opportunity of a possible speedup. PSM CLEAR provides load value prediction at the execute/memory stage which results in a simpler interface design and performance

speedup. Second, CLEAR was originally designed as a microarchitecture that needs to be hardended onto the silicon. PSM CLEAR is an implementation of the redesigned CLEAR microarchitecture on PSM substrate which intervenes the processor's pipeline stages using a well defined signal interface or PSMI.

Runahead execution [14] is a microarchitecture that retires unresolved L2 cache missed loads as they reach retirement head and marks them as invalid. It propagates invalidation to skip the execution of instructions dependent on the load. The processor exits Runahead mode when the value returns from the memory by discarding all work and rolling back the architectural state using a checkpoint, effectively warming up the caches and various predictors, thereby speeding up the overall application. PSM CLEAR differs from Runahead as it uses load value prediction, enabling it to continue execution without the need of rolling back in the case of a correct load value prediction.

1.5 Contributions of this thesis

The key contributions made in this thesis are:

- Re-design the CLEAR microarchitecture to supply load value predictions early-on in the pipeline which reduces design complexity of the PSM interface and also improves performance.
- Present a study on the peak performance of the machine using stride value predictor, oracle value predictor, and a statistical based predictor - which can be configured for a desired coverage and accuracy.
- Design of PSM CLEAR, a microarchitecture inspired from CLEAR that is implemented on the PSM substrate.
- Design of PSMI to support the PSM CLEAR microarchitecture.
- Implementation of the PSM CLEAR microarchitecture and the interface in a C++ simulator that models a conventional out-of-order core based on RISC-V ISA [15].
- Attempt the implementation of the PSMI in AnyCore [16], which is a Register-Transfer Level (RTL) design of a highly adaptive superscalar core, and map the interfaces to a module “PSM" which internally references SystemVerilog-DPI calls to implement the functionality of the PSM CLEAR microarchitecture.

1.6 Thesis Organization

The rest of this thesis delves deeper into the detailed design and analysis of the PSM CLEAR microarchitecture, and the PSMI required to support it.

Chapter 2 presents a study motivating load value prediction at execute/memory as opposed to retire and describes the different components associated with the proposed PSM CLEAR design.

Chapter 3 describes the design of PSMI that the processor needs to support to enable PSM CLEAR design. Various changes in the pipeline associated to each of these interfaces are also discussed.

Chapter 4 provides details on the experimental setup used in this work and presents results that demonstrate the advantage of the proposed PSM CLEAR + PSMI design.

Chapter 5 summarizes the contributions of this work and outlines the future work.

CHAPTER

2

CLEAR FOR PSM

Checkpointed Early Load Retirement [3] is a mechanism to unclog the active list when an unresolved load reaches its head. The technique involves checkpointing of the architectural registers and providing a load value prediction to achieve the illusion of a large window processor. The load value predictor associated with the traditional implementation of CLEAR provides a prediction only when the unresolved load reaches the retirement head. The design choice of providing load value predictions at retire causes the dependent instructions to stay in the issue queue for longer as they are woken up only when the predicted value is supplied, missing the potential of a speedup in the case of a correct prediction.

In this chapter, we present a study on how the performance of processor supporting CLEAR changes if the value prediction is provided at execute rather than at retire.

This chapter also describes a detailed design of PSM CLEAR, building blocks required to support it, and the interactions between the processor and the PSM substrate.

2.1 Load Value Prediction at EXE/MEM

Load value prediction is a mechanism which is used to predict 32/64-bit values based on the history and nature of loads. These value predictions might be injected in the front-end stage (e.g., decode, dispatch), or in the back-end stage (e.g., execute, retire), each affecting the design and performance of the overall system.

The benefit of value injection in the front-end stage is early-wakeup of dependent instructions which might reduce the pressure on issue queue and allow more instructions to execute. Applications that have high L1 hit rates might not get any benefit from such a mechanism and might incur heavy misprediction penalty. We might not want to use this mechanism as the performance penalty (squashing everything) overpowers the potential speedup (#cycles between dispatch and execute) by a significant amount.

In order to bridge the above mentioned performance gap, we propose predicting the values at the execute/memory stage, and using them only in the case of a data cache miss. The biggest advantage of this mechanism is that it significantly reduces the design complexity of both PSM interface and the processor.

Figure 2.1 shows the IPC for different configurations like baseline, baseline with perfect data cache, CLEAR (LVP at retire), CLEAR with LVP at EXE/MEM, CLEAR with infinite architectural register checkpoints (LVP at retire), CLEAR with infinite architectural register checkpoints and LVP at EXE/MEM for ASTAR application with perfect LVP support.

Table 2.1 shows the processor configuration used for the baseline. For CLEAR, only one checkpoint was used that could hold at-most 16 unresolved loads, i.e., a single misprediction amongst these 16 can cause the processor to rollback to the checkpointed state. We use this configuration throughout the text unless explicitly specified.

The reason why CLEAR + LVP at EXE/MEM performs better than perfect data cache in some cases is dependent on the simulator model of the processor pipeline. With perfect data caches, all loads are hits and are immediately moved to writeback stage from execute. Upon reaching writeback, loads are marked as complete after which they wait in the active list until the retirement head reaches their index. In the best case, it takes two cycles for a load to retire, and one cycle to wake up the dependents (in execute).

With CLEAR + LVP at EXE/MEM, a load missed in the data cache acquires a predicted value, and woken up which is similar to perfect data cache configuration. It stays in the load queue where it replays itself until it gets resolved. The unresolved load instruction is fake-retired when it reaches the head of the active list. Thus, the time to retire such a load is (#instructions before this load) / 16, and the time to wake up its dependents is one cycle.

Table 2.1: Summary of the baseline architecture modelled

Processor width	16 instr./cycle
Branch predictor configuration	**Type:** Gshare **BP table size:** 65536 **RAS size:** 32
Checkpoints for BP recovery	32
L1 D$	**Size:** 64 KB **Hit latency:** 1 cycle **Miss latency:** 10 cycles
Unified L2	**Size:** 256 KB **Hit latency:** 10 cycles **Miss latency:** 100 cycles
Active list	128/256/512/1024/2048
Issue queue	32/64/128/256/512
#Load queue; #Store queue	#Active list / 2

Time to retire the instruction becomes one cycle if the number of instructions before this load is less than or equal to 16, which is less than the perfect data cache case. This one cycle difference at retirement is the primary reason for the slightly better performance of CLEAR + LVP at EXE/MEM when compared with perfect data cache configuration.

In this work, we always use load value prediction at execute/memory stage and not at the retire stage, as there is a measurable difference in the performance. In the next section, we propose the design of PSM CLEAR with LVP at execute/memory stage.

2.2 PSM CLEAR microarchitecture

This work assumes the following two requirements before designing PSM CLEAR:

1. The signal interface or PSMI supporting it should have simple control logic and contain as few wires as possible.

2. Changes in the processor pipeline supporting such an interface should be minimal and complex structural changes must be avoided to reduce the risks and cost associated with them. Simple changes in the design might also be verification friendly.

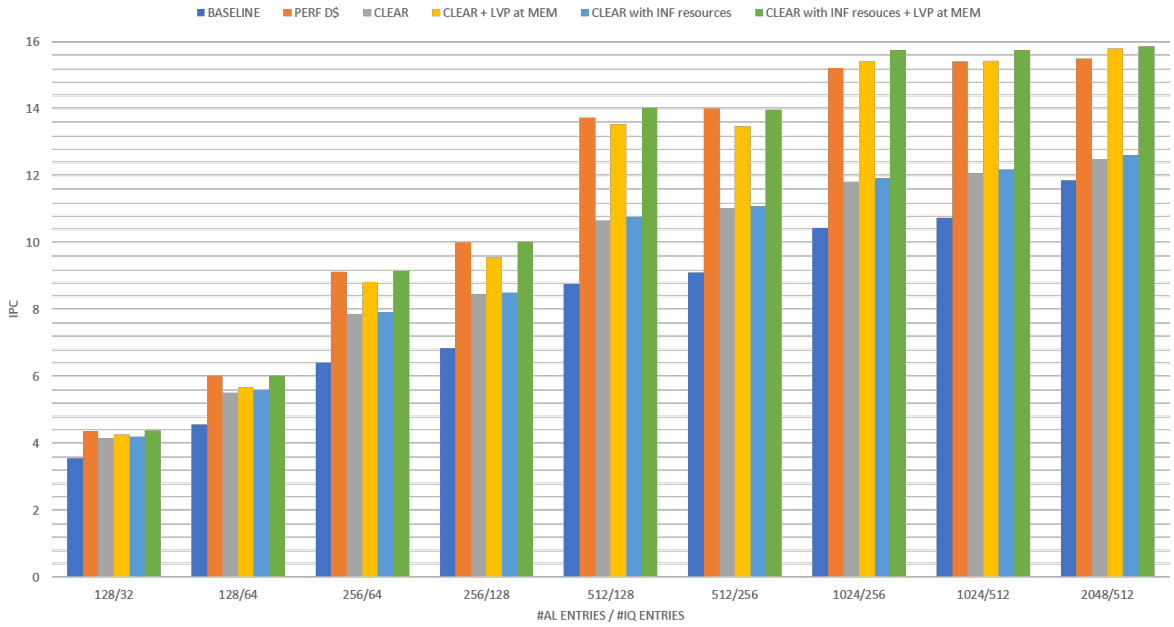

Figure 2.1: Motivation for LVP at EXE/MEM stage (ASTAR Application)

Figure 2.2 shows the high level design of the proposed PSM CLEAR microarchitecture. In order to understand the complete design, it is essential to understand the use and rationale behind each of the components used. All of the structures might impose a structural hazard, thus, limiting the performance of the overall solution. The following sections describes each of them in detail.

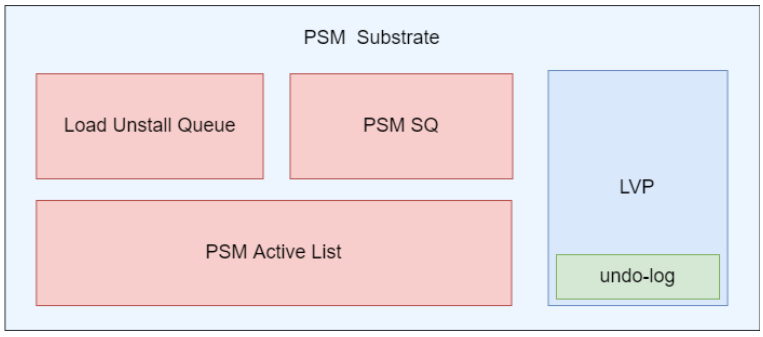

Figure 2.2: High level design of PSM CLEAR

2.2.1 PSM Active List

In the previous section, we concluded that a load value predictor if installed at execute / memory stage would perform better than the one installed at retire. Typically, load value predictors have tables that contain information required to compute the 32/64-bit values. In the actual design, it is possible that these predictors require multiple cycles to compute these values. Thus, to hide the latency of load value computation, we would want to initiate the value computation as early as possible in the pipeline (like in front-end stages) and use them in execute/memory stage.

In a conventional core, instructions execute in an out-of-order manner because of which we might want to identify them individually as they access the memory. Using only the program counter (PC) for the identification has the following two downsides.

1. It requires a 32/64-bit wide signal interface between the processor and the PSM substrate.

2. It becomes difficult to identify two or more dynamic instructions that have the same program counter (e.g. load in a loop)

To simplify the design and the interface, we use the already existing active list index to correctly identify the instructions in the dynamic instruction stream. To support this design, we need to add an interface that has $\log_2$(#active list entries) wires, reducing the cost drastically from a 32/64-bit PC based interface.

With an active list indexed design, dispatch becomes the earliest stage to start the value prediction machinery. Values need to be deposited in an active list indexed payload as they are produced. This payload can be a part of the processor's active list, or can be a part of the PSM substrate. For a processor-based payload design, additional read/write ports might be required to read the state and deposit the values. For a PSM based payload design, a shadow copy of the processor's active list might have to be maintained in the PSM substrate.

In this work, we choose a PSM based active list design as it is less intrusive and results in less changes in the processor design. Entries in the list might be light-weight in terms of storage when compared with their processor's counterpart as they only need to record information like PC, availability of the predicted value, predicted value, confidence of the predicted value, load/store type, etc.

PSM based active list might also give some slack to the value prediction machinery. A load that missed in the data cache might also have an unavailable value in the active list as

the value prediction machinery might not have completed its computation. Such loads can simply be marked as a miss and might get the prediction when replayed by the processor.

To maintain synchronization between the processor's active list and the PSM's active list, we require interventions at the following stages:

- **DISPATCH:** Instructions are dispatched in the PSM active list as they are dispatched onto the processor's active list.
- **WRITEBACK:** In case of a branch misprediction, processor rolls back its active list tail pointer in order to squash instructions post the mispredicting branch. A signal interface is required at this stage to convey the information of fixing the list.
- **RETIRE:** An entry from the active list is popped as it retires. A signal interface that gives the information on number of instructions that retire at any given cycle is sufficient to synchronize the PSM version of the active list.

2.2.2 Load Unstall Queue

When an unresolved value predicted load reaches the head of the active list, it initiates the processor to enter CLEAR mode. This is done by checkpointing the architectural registers and entering fake-retire mode. Instructions that retire from this point are retired speculatively. Thus, the PSM CLEAR needs to track all the unresolved loads that were fake-retired. True values are compared against the predicted values as they return from the memory hierarchy, giving us the information if it were a mispredict or not.

In order to track all such outstanding load instructions, a FIFO based structure known as load unstall queue is used. PSM CLEAR microarchitecture is supposed to replay all such loads until they are resolved. A load that is at the head of this buffer has the highest priority to be replayed as it is the oldest instruction in the pipeline.

Loads are popped from this queue as they are resolved. In the case of a CLEAR misprediction, this structure is squashed completely.

2.2.3 PSM Store Queue

Once the processor enters CLEAR mode, store instructions that were fake-retired are popped from the processor's store queue and flushed onto the memory. A PSM intervention is required at this interface to capture these store instructions, redirecting them to the PSM

substrate and not letting them drain in the memory hierarchy as they are speculative in nature.

A FIFO-based in-order structure called PSM store queue is used to maintain these fake-retired stores. This structure is either squashed in case of a CLEAR misprediction or drained in-order to the memory hierarchy in the case of a correct CLEAR resolution.

Multiple stores that share the same address are not coalesced in this buffer as processor consistency model is assumed.

This structure requires support of store load forwarding to provide any latest value it might have buffered. These forwarded values have a higher priority than the values from load value prediction machinery. Thus, if a missed load has an address match in the PSM store queue, it is forwarded a value from the store queue entry and not from the prediction machinery. All stores in this structure will have known store addresses resulting in a definitive memory disambiguation.

Partial store matches happen where there is no complete overlap between the range of addresses requested in a load and the range of addresses available in a store queue entry. Multiple store queue entries might collectively have the complete data, but, stitching them together might result in a complicated hardware design. In our design, we neither attempt to stitch multiple store queue entries, nor provide a partial value as that might be incorrect. Also, as a part of the design choice, we do not use a predicted value in this particular case. As no value is provided in the case of a partial store match, it becomes a cache miss from a processor's point of view.

Figure 2.3 shows the four different cases of store-load forwarding. In order to complete the combinations, figure 2.4 shows two invalid cases which will never happen due to address misalignment.

2.2.4 Load Value Predictor

Design of an efficient load value predictor is essential in the PSM CLEAR design space. Value mispredictions that are resolved before they enter CLEAR mode (i.e., before they reach the head of the active list) are less expensive as compared to the CLEARed loads due to the cost of recovering the architectural register state associated with it.

In our implementation, we only have one CLEAR checkpoint that can record at most 16 unresolved value predicted loads, the cost of a misprediction becomes very high as a single misprediction amongst these 16 will initiate a recovery.

In this work, we use the following three predictors:

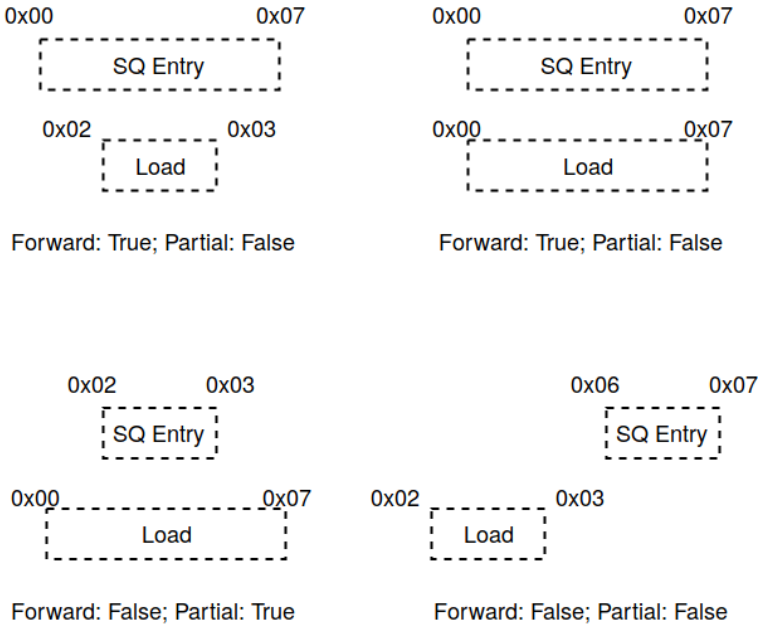

Figure 2.3: Valid cases of store-load forwarding

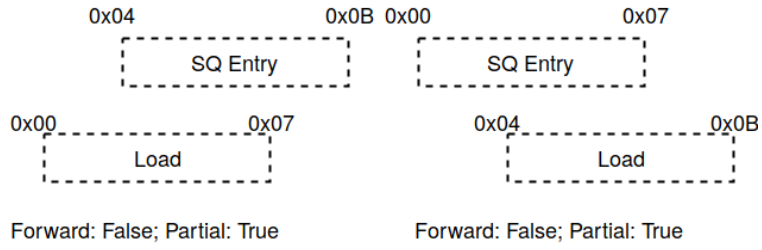

Figure 2.4: Invalid cases of store-load forwarding due to address misalignment

- **Oracle value predictor:** This predictor uses oracle values that we grab from the functional simulator. This was implemented to facilitate limit studies in the PSM CLEAR design space.

- **Statistical predictor:** This predictor relies on random numbers to decide if the value predicted is an oracle value or an incorrect value and can be configured to achieve a desired load coverage and accuracy. This was also implemented to facilitate limit studies and is not a real predictor.

- **Stride value predictor:** This predictor [11] has a PC indexed table that record a base, stride pair and number of inflight instructions corresponding to a PC to compute a value prediction. In our implementation, we assume infinite sized tables which removes any aliasing effects that tables of fixed size might incur. The predictions are also guarded by a confidence mechanism which is implemented using a saturating counter present in the table entries.

In our implementation, all of the above mentioned predictors take a single cycle to compute the values.

We update the load value prediction machinery at retire. Thus, base, stride, and confidence counters are updated as the instructions retire/fake-retire. For fake retired instructions, LVP tables are updated speculatively and might need recovery. Although, LVP is a part of microarchitecture and recovering it is not needed functionally, we still might want to recover for performance reasons.

LVP undo-log

LVP undo-log is a FIFO based structure that is required to facilitate recovery of the LVP tables if clobbered in the fake-retired mode. In the case of a CLEAR misprediction, all fake-updated values need to be rolled back to a state prior to entering the CLEAR mode. Thus, the undo-log only needs to record values that were clobbered in the CLEAR mode. To support such a mechanism, each entry in the LVP table has a clobbered bit which is reset before entering CLEAR mode.

When a load instruction fake-retires in CLEAR mode, it first checks for the clobbered bit. If the bit was unset, current values of the base, stride and confidence counters are pushed to the undo-log along with the index to the entry in the LVP table (LVP_index). The clobbered bit is set once the base, stride and confidence counter values are updated/clobbered.

In the case of a recovery, entries are popped from the undo-log to restore the LVP tables. LVP table entries that are restored can be indexed using the LVP_index that was recorded when an entry was pushed onto the undo-log. Undo-log is squashed in the case of correct CLEAR mode resolution.

It is important to note that the support for undo-log was possible as the LVP tables were a part of the PSM substrate. Similar argument for processor's global history register, RAS, etc., cannot be made as they might require dedicated interfaces to read/write which might be expensive to support.

2.2.5 Architectural Register Checkpoint

The simulator/AnyCore infrastructure we used in this work has a physical register file (PRF) which contains both the architectural and speculative registers. This reduces the overhead on committing and freeing registers as no data movement is involved. A Renaming Map Table (RMT) is used to map the speculative state mappings of the logical and physical register. An Architectural Map Table (AMT) is used to map the architectural state mappings of the logical and physical registers.

The benefit of a PRF based design is that the rename stage provides a tag to identify the physical registers, reducing the number of bits transferred in each stage as opposed to data value (32/64-bit) movement. Also, register read stage can be located after issue and before execute stage improving the locality and placement from a physical design standpoint.

All physical register in the PRF can be mapped to one of the following three states:

- **Architectural:** These registers are a part of the processor's architectural state. At any given time, there will always be a fixed number of registers in this state (#integer architectural registers + #floating architectural registers) and these registers will have a valid mapping in the AMT.

- **Speculative:** These registers are a part of the speculative state and may or may not have a valid mapping in the RMT. Registers in this state are promoted to the architectural state when an instruction with a valid destination mapping retires. At this point, a register which was a part of the architectural state is added to the list of free registers.

- **Free:** Register that are neither architectural nor speculative are a part of the free state. When required, the renaming machinery can promote a register from free state to speculative state.

A FIFO based free list is used in this design to track the available free registers at any given time in the pipeline. Registers are popped from the tail, promoting them to the speculative state. Registers pushed onto this list are moved from the architectural state to the free state.

At retire, current mapping in the AMT is pushed onto the free list. Such a register might be mapped to a destination register for a younger instruction which might clobber it. This operation is correct and expected in normal mode. In CLEAR mode, the instructions retire speculatively and hence need register checkpointing to enable recovery to the architectural state in the case of a misprediction.

Such a checkpoint can be supported in multiple ways:

1. Keep an up-to-date shadow copy of the architectural registers during the normal operation. Freezing the updates to the shadow copy during CLEAR mode can be used to create the checkpoint.

2. The processor can explicitly copy the architectural register values in a storage at the time of checkpointing.

3. The processor can checkpoint the AMT and pin the physical registers to disable their movement onto the free list. This option reduces the number of free registers for the lifetime of the checkpoint.

In our implementation, we use (3) to checkpoint the architectural register. The design of the AMT and retirement stage needs to support an additional pinned bit. These bits are flash set when the processor enters CLEAR mode. As an instruction retires, the processor checks for the pinned bit in the AMT corresponding to the destination logical register. If the bit is found to be set, the processor doesn't push the physical register onto the free list and simply updates itself with a new mapping. For both normal and CLEAR mode, the processor can simply reset the bit autonomously at retire if the destination register is valid.

If the CLEAR mode resolves to be a correct prediction, pinned entries that were updated in the AMT needs to be added to the free list. This can be implemented by iterating through the AMT and pushing the mappings from the checkpointed AMT to the free list if the pinned bit is found to be unset.

In the case of a CLEAR misprediction, entries that were updated in the AMT needs to be added to the free list. This can be implemented by iterating through the AMT and pushing mappings from the AMT to the free list if the pinned bit is found to be unset.

Pinned bits are flash cleared before the processor enters normal mode of operation and is not dependent on the nature of CLEAR mode resolution.

2.2.6 AMT Checkpoint

Architectural Map Table (AMT) is a structure that records the logical register to physical register mappings and represents the architectural state of the processor. Entries in the AMT are updated only when they become a part of the architectural state, hence, at retire.

In CLEAR mode, instructions are fake-retired and might poison the state of Architectural Map Table. As a part of the CLEAR architectural state checkpoint, AMT needs to be backed-up. In the case of a correct CLEAR resolution, the CLEAR checkpoint can simply be released. However, in the case of a misprediction, processor's AMT needs to be restored from version in the CLEAR checkpoint.

The location of the AMT checkpoint can either be in the processor pipeline or in the PSM substrate. As a part of the PSM interface design, a processor might support one checkpoint to expedite the process of backup and restore which might increase the performance. Implementing the AMT checkpoint in processor's substrate might also allow for a flash (single cycle) backup/restore by adding a flip-flop based or a custom SRAM design. Other implementations might require additional read/write ports for the backup/restore process.

In our implementation, we assume that the AMT checkpoint is a part of the processor microarchitecture and supports flash backup/restore. Signal interface required to initiate these processes are thus a part of the PSMI.

CHAPTER

3

CHANGES IN THE PROCESSOR

In the previous chapter, we had emphasized on the PSM side of the microarchitecture. In this chapter, we will discuss on the interface support required from the processor and the associated changes in the processor pipeline design.

Figure 3.1 shows the different stages in the baseline processor pipeline. The following section improves upon these pipeline stages to add support for PSMI. Figure 3.2 shows a high-level representation of the processor pipeline and PSM substrate after incorporating the PSM interfaces.

3.1 PSM CLEAR Interface support

By now, we have designed the PSM CLEAR microarchitecture and identified various components associated with it. In this section, we provide a detailed design of the interfaces that are required to support the proposed PSM CLEAR microarchitecture.

It needs to be noted that the interfaces were made in accordance with the design implemented in AnyCore [16]. The following sections provide details on the interfaces split across different stages in the processor pipeline.

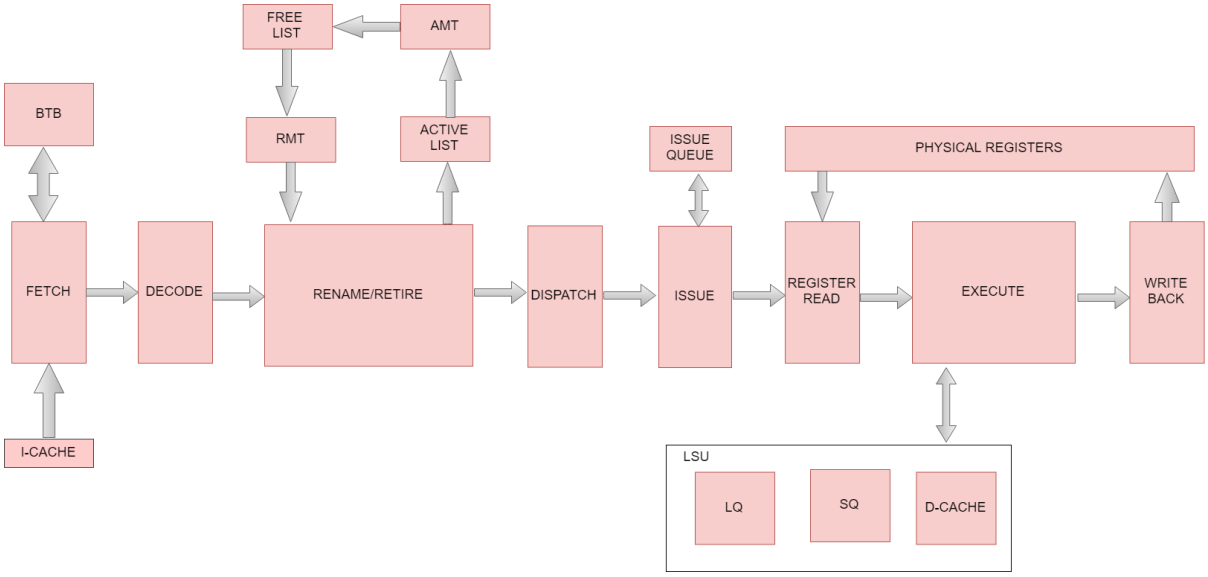

Figure 3.1: Processor pipeline in the baseline implementation

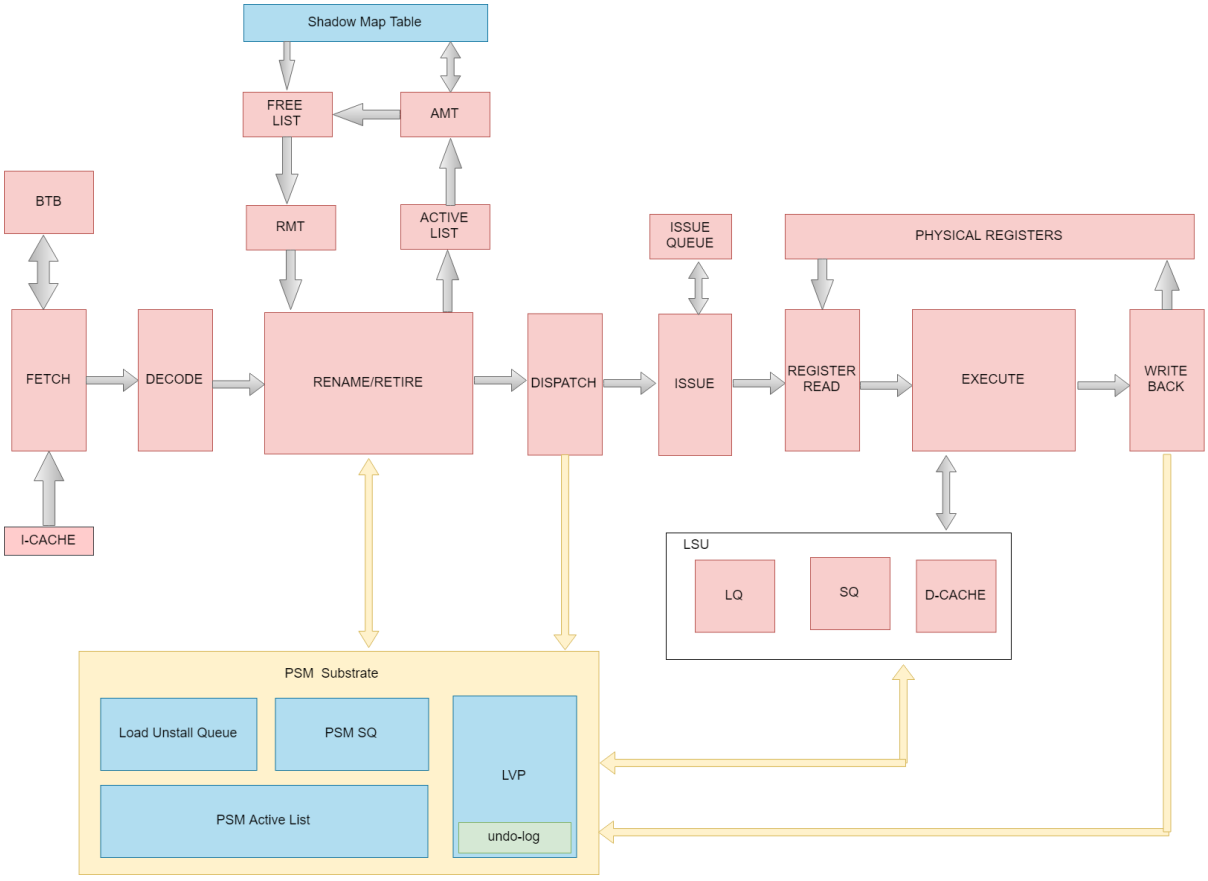

Figure 3.2: Processor pipeline with PSM CLEAR

3.1.1 Dispatch

We need the following two signals from the dispatch stage. The direction of both the signals is from processor to PSM CLEAR:

1. **Ready:** This is a single bit signal that is asserted if there is at least one valid instruction to be dispatched in that cycle.

2. **AL Packet:** Active list packets (#dispatch width) are created in the dispatch stage and contain multiple fields like valid, PC, logical destination register, physical destination register, instruction type, etc. For our PSM CLEAR design, valid bits need to be accounted for the packet validity, PC for the LVP machinery indexing, and is_load to track load instructions.

3.1.2 Execute/Memory (LSU)

The LSU pipeline in AnyCore can issue one load and one store per cycle to the memory subsystem.

A load requested to the memory can either be a packet from the AGEN stage, or a replay packet waiting to be resolved. Higher priority is given to AGEN packets over replay packets as they win memory lane arbitration for that cycle. Following are the signals required to support the load data path:

1. **Load Packet:** This packet contains information required to perform a load operation and includes fields like valid, address, size, sign information, active list index, and PSM metadata - use of which is described later in this section. If a load packet is valid, active list index is provided to the PSM CLEAR which returns the following signals:

    - **PSM Hit:** This signal interface is asserted if the PSM CLEAR data path had a hit which could be due to the availability of a store-forwarded value or due to a confident load value prediction. An additional bit might be required to signal the processor for a store-load forward which might override the data supplied from the data cache.
    - **PSM Data:** This is a 32/64-bit signal interface that provides value to the load in the processor pipeline from PSM CLEAR.

The following signals are required to support the store data path:

1. **Store PUT:** A single bit interface that signals the processor to redirect a store to the PSM substrate instead of data cache. PSM CLEAR asserts this signal while in CLEAR mode.

2. **Store GET:** A single bit interface that informs the processor to redirect a store packet from the PSM substrate to the data cache. This signal is asserted by the PSM CLEAR for multiple cycles after detecting a correct CLEAR mode resolution to drain all the buffered stores to the memory.

3. **Store Packet PUT:** This interface contains store packets redirected from the data cache to the PSM substrate when the Store PUT interface is asserted. Store packets have a valid bit along with address and data information.

4. **Store Packet GET:** This interface contains store packets redirected from the PSM substrate to the data cache when the Store GET interface is asserted. Store packets have a valid bit along with address and data information.

Value predicted unresolved loads need to be replayed from the PSM substrate as they are hits from a processor point of view, thus, require the following interface:

1. **Replay Packet:** This is similar to the load packet interface previously described with the difference in the direction. Replay packets are generated in the PSM substrate and injected onto the processor pipeline where they compete against AGEN packet, and the replay packet from the processor. A replay packet will eventually arrive at the Load Packet interface if the processor selects it.

In order to identify the origin of load packets, we extend the structure of a load packet to include a field called PSM metadata. We use this field to identify if the the originator of a particular packet was a processor, or PSM substrate. PSM CLEAR can populate this field to identify if it was originated from the PSM active list, or PSM load unstall queue. Similarly, PSM CLEAR can update its structures with the data cache value as they return from the memory.

3.1.3 Writeback

Branch mispredictions that are identified in the writeback stage might initiate recovery of multiple structures including active list. Following are the signal interfaces that are required at this stage:

1. **AL index:** This contains the recovery index which is used to update the tail pointer of the PSM active list in the case of a misprediction recovery.

2. **Fix List:** This is a single bit interface which if asserted causes PSM CLEAR to update its active list using the AL index from the previous interface.

In our work, writeback interface was only required in the C++ simulator (721sim) as AnyCore recovers from branch misprediction using a full squash technique, which waits for the mispredicted branch to reach the retirement head.

3.1.4 Retire

Following signal interfaces are required at the retire stage:

1. **Total commit:** This is the number of instructions that are retired in a given cycle. This interface is used to update the PSM active list by popping “Total commit" instructions from its head.

2. **Exception flag:** This is a single bit interface which if asserted from the processor informs the PSM CLEAR to full-squash its active list as the instruction at the retirement head might have encountered an exception.

3. **Misprediction flag:** This is a single bit interface which if asserted from the processor informs the PSM CLEAR to recover its active list as the instruction at the retirement head might have encountered a branch misprediction. In the AnyCore design, this results in a full-squash

4. **Violation flag:** This is a single bit interface which if asserted from the processor informs PSM CLEAR to full-squash its active list as the instruction at the retirement head might have encountered a load violation due to speculative memory disambiguation.

5. **Checkpoint AMT:** This is a single bit interface which if asserted from PSM CLEAR triggers the processor to backup up the AMT onto a shadow map using flash copy. This is asserted when the processor enters CLEAR mode.

6. **Restore AMT:** This is a single bit interface which if asserted from PSM CLEAR triggers the processor to restore the AMT from the shadow map using flash copy. This is asserted only in the case of CLEAR misprediction.

7. **Set AMT pin:** This is a single bit interface which if asserted from PSM CLEAR, flash sets the pinned bits in the processor's AMT.

8. **Clear AMT pin:** This is a single bit interface which if asserted from PSM CLEAR, flash clears the pinned bits in the processor's AMT.

9. **Free registers:** This is a single bit interface which if asserted from PSM CLEAR, adds registers to the free list if the pinned bits in the AMT were unset. Free register source interface described below is used to select between the AMT and shadow map table to read a physical register. As this operation takes a few cycles (#AMT / commit width), the retirement engine stalls as the write ports to the free-list are hijacked. PSM CLEAR asserts this signal in the case of a correct CLEAR mode resolution.

10. **Free register source:** In the case of a correct CLEAR resolution, checkpointed registers that were overwritten in the AMT needs to be freed (select shadow map table in this case). For CLEAR misprediction, poisoned AMT entries needs to be freed as they were speculative (select AMT in this case). This bit selects between the AMT and shadow map table.

11. **Free register ack:** This is a single bit interface which is asserted by the processor when the Free Registers operation completes and acts like a handshaking mechanism between the processor and PSM CLEAR.

12. **Stall Commit:** This is a single bit interface which if asserted from PSM CLEAR stalls commits in the processor. This is required to support structural hazards in the PSM CLEAR implementation.

13. **Recovery PC:** This is a 64-bit bus from PSM CLEAR to processor that is used to correct next PC logic in the case of a CLEAR mode misprediction.

14. **Squash:** This is a 2-bit signal interface from PSM CLEAR to processor. Squash[0] if asserted squashes the processor's pipeline. Squash[1] if asserted selects the recovery PC provided from the PSM CLEAR using the above interface.

CHAPTER

4

EXPERIMENTAL SETUP

In this chapter, we discuss the simulated environment and the applications used for evaluating our PSM CLEAR design.

4.1 Simulated Architecture

We evaluate our design using 721sim, a detailed cycle-level execute-at-execute execution-driven C++ superscalar processor simulator used in Prof. Rotenberg's research lab and ECE 721 Advanced Microarchitecture course. It is based on RISC-V ISA [15] and has a detailed structural model of an out-of-order processor and its memory subsystem. The model supports multiple configurations like variable fetch, dispatch, issue, and retire width. It also supports different oracle modes like perfect fetch, perfect data caches, perfect branch prediction, etc.

The simulator models speculative memory disambiguation but lacks a load violation predictor [17]. The simulator model also lacks a prefetcher of any sorts. The baseline processor is a four-wide dynamic superscalar core that has two levels of on-chip caches. The architectural parameters are listed in the table 4.1

The simulator models Fetch, Decode, Rename and Dispatch as the front-end pipeline stages. Because of the intervening Fetch Queue, the front-end supports two separately specified widths i.e., fetch_width and dispatch_width. Rename stage is sub-pipeline into Rename1 and Rename2. Rename1 tries to obtain a full rename bundle (dispatch_width) from the fetch queue. Rename2 renames the rename bundle. Figure 4.1 describes the front-end pipeline stages.

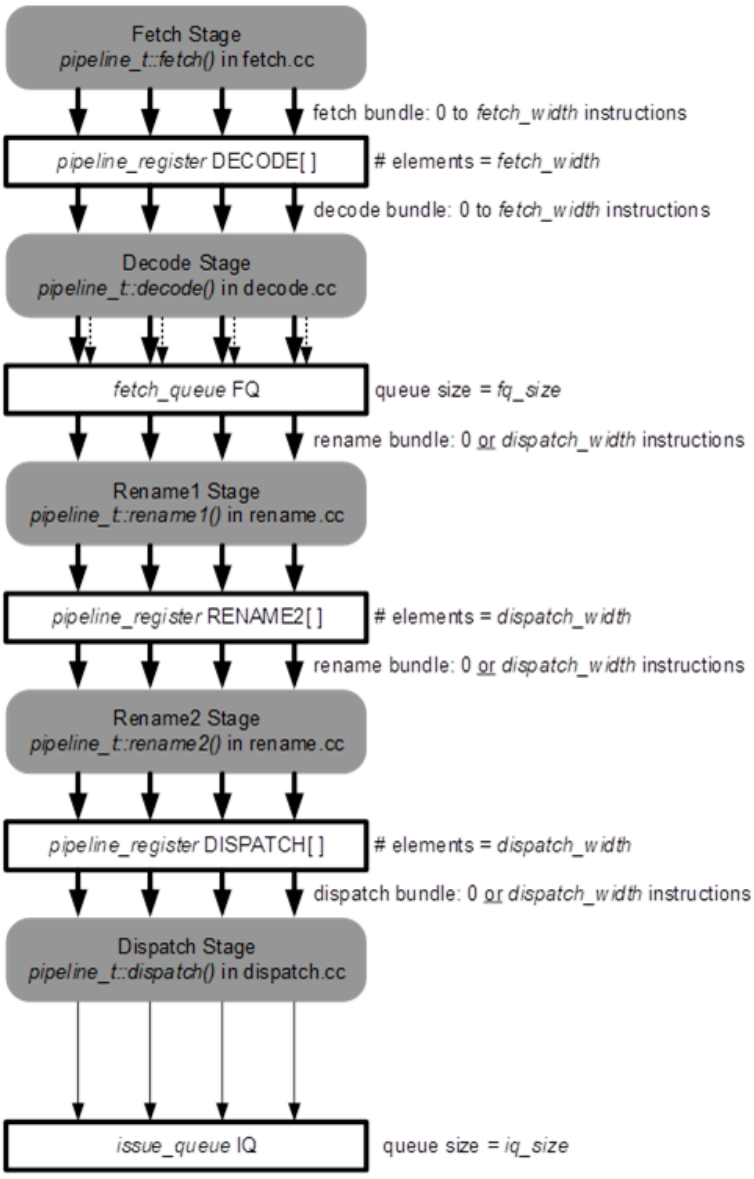

Figure 4.1: Description of the front-end stages: Fetch, Decode, Rename and Dispatch [1]

The simulator models Issue, Register Read, Execute and Writeback as the back-end pipeline stages. The select and issue logic selects upto issue_width ready instructions per cycle from the issue queue which are sent for execution. An instruction leaves the pipeline after the Writeback stage and occupies entries in the active list, load and store queues and branch predictor queue until it retires. Figure 4.2 describes the back-end pipeline stages.

In our implementation, we assume that both the processor and the PSM substrate operate at the same clock.

Table 4.1: Processor configuration

Fetch width	4
Issue width	8
Retire width	4
Branch predictor configuration	**Type:** Gshare **BP table size:** 65536 **RAS size:** 32
Checkpoints for BP recovery	8
L1 D$	**Size:** 64 KB **Hit latency:** 1 cycle **Miss latency:** 10 cycles
Unified L2	**Size:** 256 KB **Hit latency:** 10 cycles **Miss latency:** 100 cycles
Active list	128
Issue queue	32
#Load queue	64

4.2 Applications

We evaluate our mechanism using 5 integer benchmarks from SPEC2006 [18] suite and 2 integer benchmarks from SPEC2017 [19] suite. Details on each application can be found in the table 4.2. We used the SimPoint3.2 tool [20] to identify the top weighted 100 million instruction region for simulation with *test* input sets.

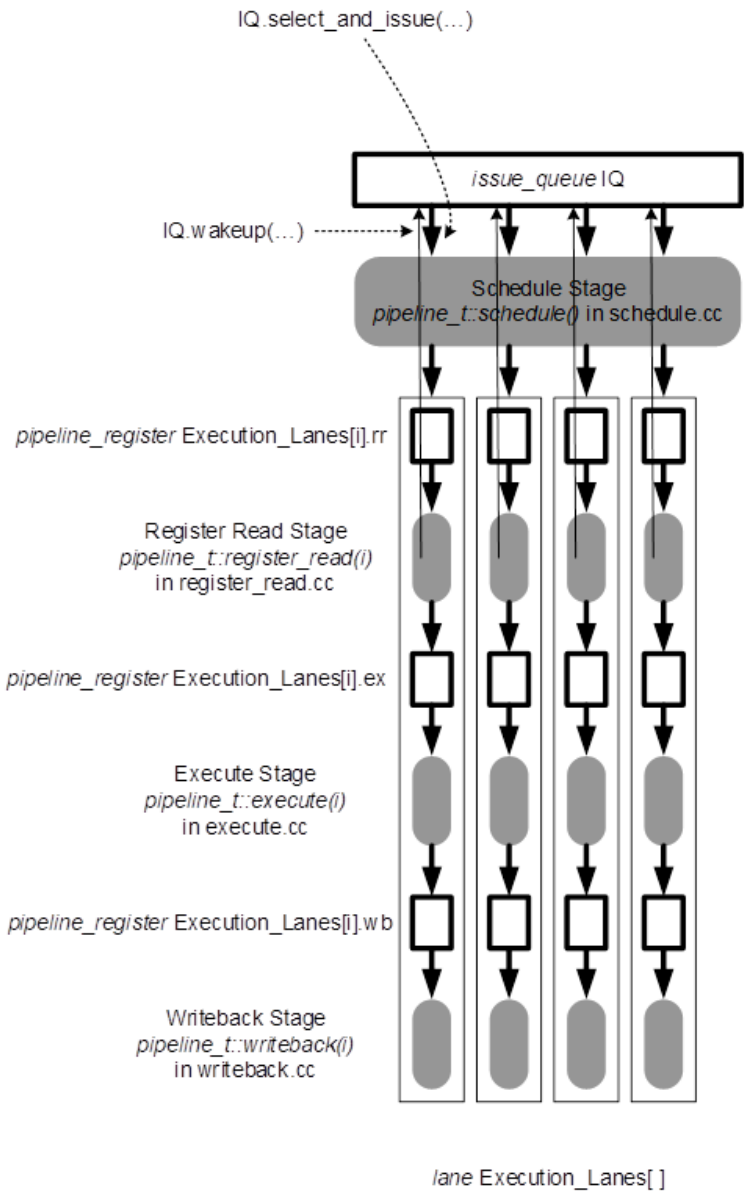

Figure 4.2: Description of the back-end stages: Issue, Register Read, Execute and Writeback [1]

Table 4.2: Details on applications from SPEC2006, SPEC2017 suite

Application	**Suite**	**Brief Description**
473.astar	SPEC2006	Pathfinding library for 2D maps, including the well known A* algorithm
401.bzip2	SPEC2006	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.
456.hmmer	SPEC2006	Protein sequence analysis using profile hidden Markov models (profile HMMs)
429.mcf	SPEC2006	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
458.sjeng	SPEC2006	A highly-ranked chess program that also plays several chess variants.
600.perlbench_s	SPEC2017	A cut-down version of Perl v5.22.1, the popular scripting language with most of OS-specific features removed.
657.xz_s	SPEC2017	Based on Lasse Collin's XZ Utils 5.0.5, with a few differences.

4.3 Results

4.3.1 Stride-LVP Performance

In this work, we use load coverage and LVP accuracy to quantify the performance of our stride based load value predictor [11]. The performance counters are measured only for correct path instructions as they retire.

We categorize the LVP statistics into 4 bins with which both coverage and accuracy can be derived:

1. **MISPRED-UNCONF:** This represents the fraction of retired loads that would have been a mispredict and were not predicted due to low confidence.
2. **MISPRED-CONF:** This represents the fraction of retired loads that were mispredicts and were predicted as they had high confidence. This fraction must be minimized to reduce misprediction penalty.
3. **CORR-UNCONF:** This represents the fraction of retired loads that would have been correct predictions and were not predicted due to low confidence. This fraction must be minimized as it represents lost opportunity.
4. **CORR-CONF:** This represents the fraction of retired loads that had correct predictions and were predicted due to high confidence. This fraction must be maximized to exploit the full potential of PSM CLEAR.

In our implementation, we use an infinite sized PC indexed table that records base, stride and confidence values to make a prediction. The values in the table are updated as they retire, hence, in-order updation of LVP machinery. We use the following two configurations of confidence counters.

1. **Low Counter:** This configuration uses a 2-bit saturating counter that increments by 1 in the case of a correct prediction and decrements by 1 in case of a misprediction. A prediction is marked confident if the counter value is greater than or equal to 2.
2. **High Counter:** This configuration uses a 4-bit saturating counter that increments by 1 in the case of a correct prediction and decrements by 3 in case of a misprediction. A prediction is marked confident if the counter value is greater than or equal to 13.

Figure 4.4 and 4.3 shows the statistics of bins for both normal mode and CLEAR mode across different applications. For both of the counter configurations, bzip has the highest fraction in CLEAR mode (approx. 46%) whereas other applications have less than 2.5%. bzip has a large fraction for the MISPRED-UNCONF bin in CLEAR mode. Reducing the fraction might require exploring more sophisticated load value prediction algorithms.

A big fraction of CORR-CONF in normal mode indicates that most of the correct load value predictions were resolved before they reached the head of active list. Such loads might typically be L1 cache misses. Applications having a high fraction of CORR-CONF bin will get performance improvements equivalent to that of hiding L1 miss latency.

The high counter configuration has a lower fraction of MISPRED-UNCONF bin whereas the low counter configuration has both, low fraction of CORR-UNCONF and high fraction of CORR-CONF bins. Overall it seems that the low counter configuration might have a better performance over the high counter configuration.

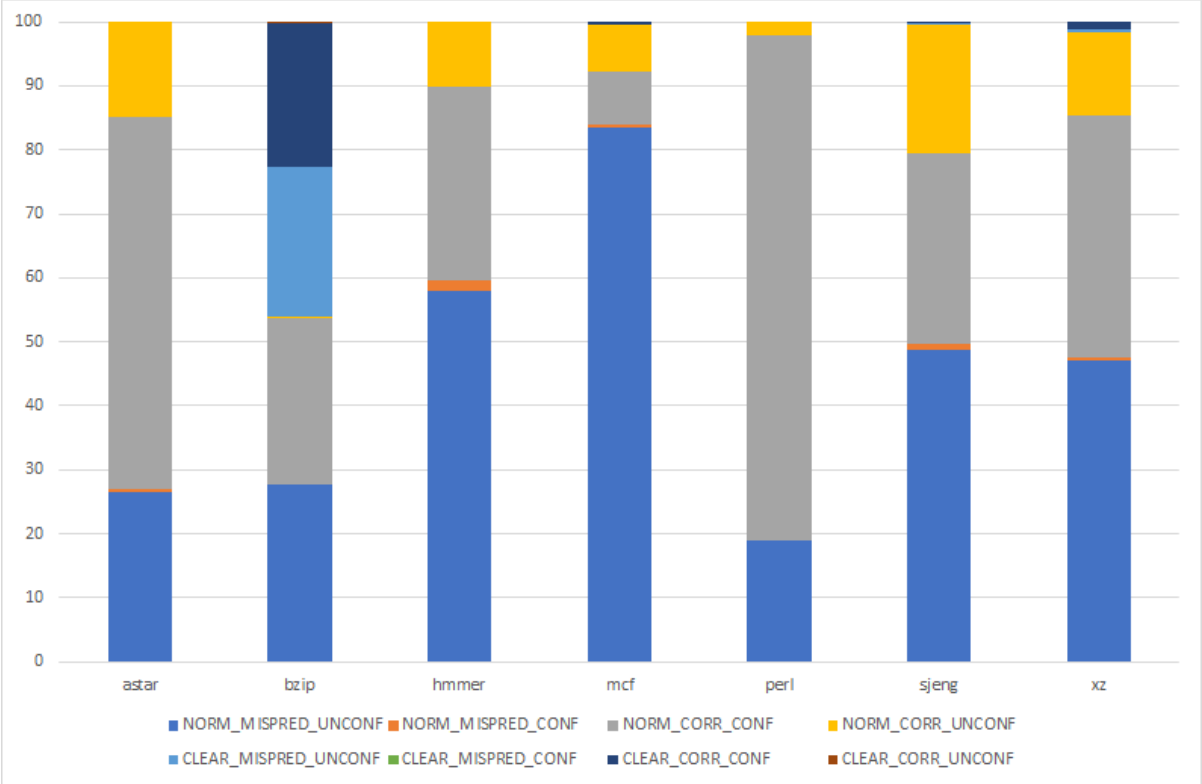

Figure 4.3: Bins for normal mode and CLEAR mode with high counters

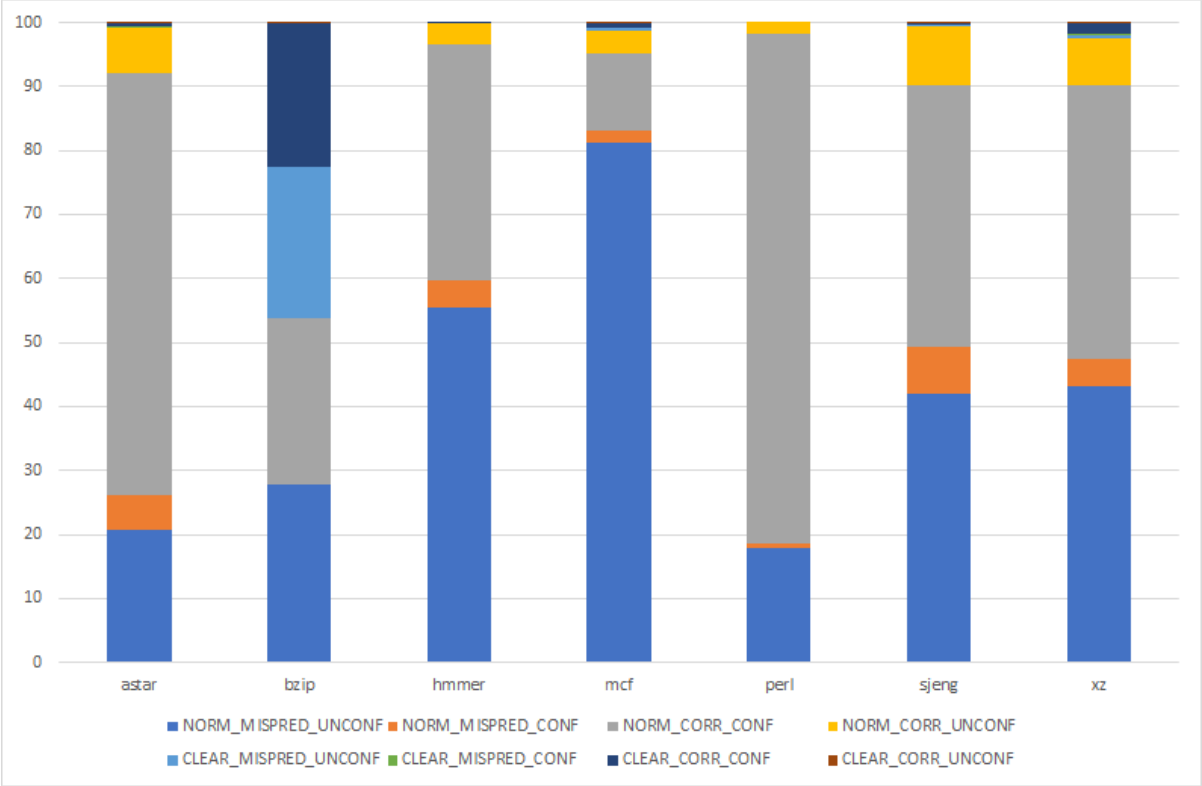

Figure 4.4: Bins for normal mode and CLEAR mode with low counters

4.3.2 PSM CLEAR

Figure 4.5 compares the performance attained by baseline, baseline with perfect data cache, baseline with perfect LVP (retire stalls till the values return), PSM CLEAR with perfect LVP support, and PSM CLEAR with different configurations of stride-LVP. The results are represented as speedups relative to the baseline.

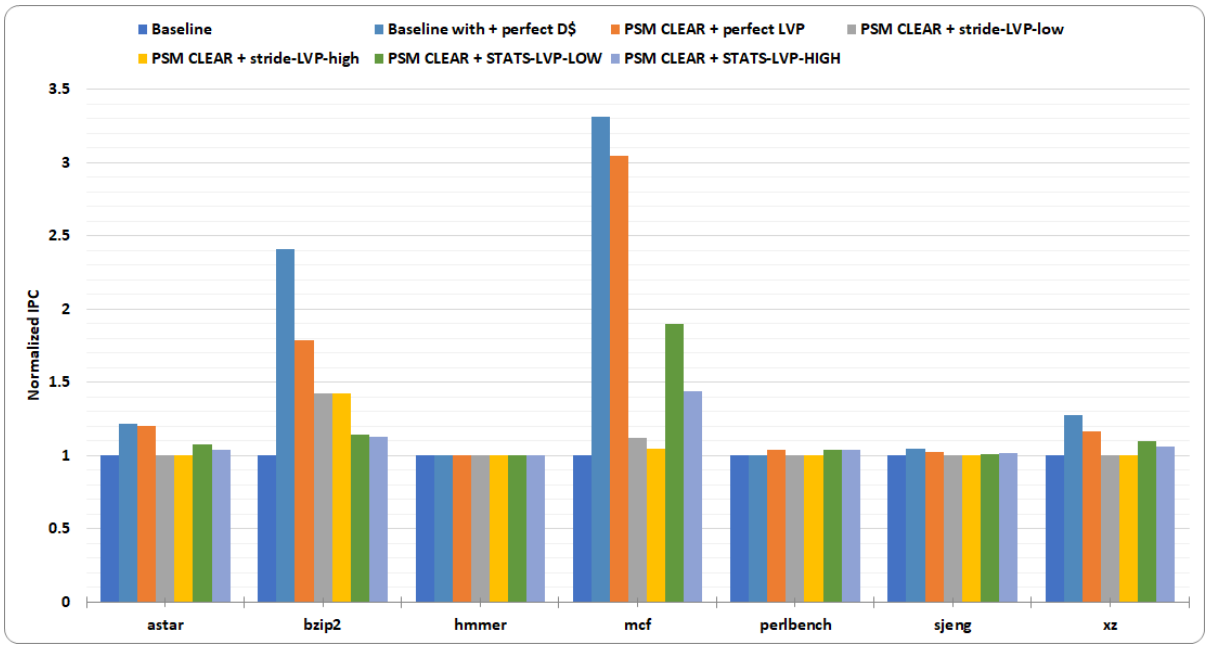

Figure 4.5: Performance of baseline, perfect data cache, and PSM CLEAR with different LVP configurations; All performance figures are speedups relative to baseline

Baseline with perfect D$ refers to the peak performance that the machine can achieve (with a caveat explained in section 2.1) and shows the performance limits to the PSM CLEAR. PSM CLEAR with perfect LVP refers to the peak performance that the PSM CLEAR microarchitecture can achieve assuming an oracle load value predictor, hence, no misprediction penalty.

PSM CLEAR with stride LVP refers to the performance attained by PSM CLEAR on using a realistic LVP machinery. The performance of the stride LVP was discussed in the previous section.

PSM CLEAR with STATS LVP refers to the performance attained by PSM CLEAR on using an unrealistic statistical predictor. A statistical predictor can be configured to achieve a

desired coverage and accuracy based on random numbers and oracle knowledge. Such a predictor is used to set the motivation on exploring different LVP mechanisms and forecast the performance of the overall system if such a predictor is replaced with the existing stride value predictor. STATS LOW refers to the system where we predict 80% of the loads with an accuracy of 95%. STATS HIGH refers to the system where we predict 50% of the loads with an accuracy of 99%

The geometric mean of the speedups for perfect D$, PSM CLEAR + perfect LVP, PSM CLEAR + stride LVP low, PSM CLEAR + stride LVP high, PSM CLEAR + STATS LVP low, PSM CLEAR + STATS LVP high are 44.31%, 35.12%, 7.05%, 6.1%, 15.38%, and 9.78% respectively. These numbers might be optimistic as the baseline simulator lacks the support of a prefetcher.

bzip and mcf being memory intensive in nature have the highest speedups with PSM CLEAR. With perfect LVP, they achieve a speedup of 78.9% and 304.8% respectively. bzip is able to achieve 42% speedup with our stride value predictor whereas mcf only achieves 12.1% for low counter and 4.8% for high counter mode. Thus, mcf requires a more sophisticated predictor to unlock the potential of PSM CLEAR. STATS low and STATS high for mcf achieves a speedup of 90.2% and 43.9% which hints us that mcf requires a predictor which has a high load value coverage.

CHAPTER

5

SUMMARY AND FUTURE WORK

5.1 Summary

In this work, we explore the design and implementation of CLEAR, a microarchitectural mechanism based on load value prediction and architectural state checkpointing. When a long latency unresolved miss reaches the head of the active list, the processor enters CLEAR mode by taking an architectural checkpoint after which it continues to fake-retire instructions until the value of the unresolved load returns from the memory. In the case of a misprediction, processor rolls back to the checkpointed state and resumes to the normal mode of operation. Even though the CLEAR mode was a mispredict, instructions executed might have had prefetched a few cache lines which might result in an overall speedup. On a correct prediction, execution in CLEAR mode is deemed correct and the checkpoint is released.

We proposed a redesigned version of CLEAR for the PSM substrate that transparently provides predicted data values to the processor by intervening its signal interface to the data cache. The design also implements structures like store queue to not let the stores drain to the memory in CLEAR mode, load unstall queue to track outstanding load misses,

active list to reduce the cost of indexing the predicted values and tracking the order of instructions. We also propose a signal interface or PSMI that the processor needs to support for PSM CLEAR to function.

Detailed simulations of our proposed design reveal a 7.05% geometric mean speedup with realistic stride load value prediction machinery. We find that applications like mcf have an opportunity for a speedup but are being bottlenecked by the algorithm of LVP machinery we used. We also provide a detailed report on the coverage and accuracy of the stride load value prediction machinery we used for both normal and CLEAR mode instructions.

We conclude that the PSM paradigm has a lot of opportunity to get application specific speedups which are limited by the PSM interface design and the underlying microarchitecture that it implements. For a PSM interface design to be successful, the impact of all the design trade offs like changes in processor pipeline, granularity of interventions must be considered.

5.2 Future Work

- Complete the implementation of all PSM CLEAR interfaces in the AnyCore design.
- Explore different load value prediction machineries and confidence mechanisms to maximize PSM CLEAR applicability.
- Replace the PSM CLEAR SystemVerilog-DPI calls in AnyCore with a High Level Synthesized (HLS) model.

REFERENCES

[1] "Ece721 - advanced microarchitecture." `https://wolfware.ncsu.edu/courses/details/?sis_id=SIS:2018:1:1:ECE:721:001`.

[2] E. Rotenberg, "FoMR: Post silicon micro-architecture, NSF grant no. CCF-1823517," 2018.

[3] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez, "Checkpointed early load retirement," in *11th International Symposium on High-Performance Computer Architecture*, pp. 16–27, Feb 2005.

[4] M. Burtscher and B. G. Zorn, "Hybrid load-value predictors," *IEEE Trans. Comput.*, vol. 51, pp. 759–774, July 2002.

[5] B. Calder, P. Feller, and A. Eustace, "Value profiling and optimization," *J. Instruction-Level Parallelism*, vol. 1, 1999.

[6] B. Calder and G. Reinman, "A comparative survey of load speculation architectures," *Journal of Instruction-Level Parallelism*, vol. 2, p. 2000, 2000.

[7] B. Calder, G. Reinman, and D. M. Tullsen, "Selective value prediction," *SIGARCH Comput. Archit. News*, vol. 27, pp. 64–74, May 1999.

[8] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pp. 226–237, Dec 1996.

[9] M. H. Lipasti, *Value Locality and Speculative Execution*. PhD thesis, Pittsburgh, PA, USA, 1998. UMI Order No. GAX98-06874.

[10] M. Lipasti, C. Wilkerson, and J. Paul Shen, "Value locality and load value prediction.," vol. 30, pp. 138–147, 09 1996.

[11] Y. Sazeides and J. E. Smith, "The predictability of data values," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, pp. 248–258, Dec 1997.

[12] M. A. Watkins and D. H. Albonesi, "Remap: A reconfigurable heterogeneous multicore architecture," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 497–508, IEEE Computer Society, 2010.

[13] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: A modular reconfigurable architecture," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, (New York, NY, USA), pp. 161–171, ACM, 2000.

[14] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *HPCA*, 2003.

[15] in *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version2.2", Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, May 2017*.

[16] R. B. R. Chowdhury, A. K. Kannepalli, S. Ku, and E. Rotenberg, "Anycore: A synthesizable rtl model for exploring and fabricating adaptive superscalar cores," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 214–224, April 2016.

[17] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, (Washington, DC, USA), pp. 142–153, IEEE Computer Society, 1998.

[18] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sept. 2006.

[19] "Spec cpu2017 benchmark." `https://www.spec.org/cpu2017/press/release.html`.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 45–57, Oct. 2002.