Formal Analysis For Processor With Memory Consistency Ordering Relaxation Support

Type	Instruction	Description
Immediate	$\langle$ **immed** $\mathsf{r_d}, \mathsf{k}$ $\rangle$	Moves integer k into register
Arithmetic	$\langle$ **op** $\mathsf{r_d}, \mathsf{r_a}, \mathsf{r_b}\rangle$	Arithmetic
Branch	$\langle$ **op** r_c, r_d $\rangle$	Branches to a different pc
Load	$\langle$ **load** r_d, r_a $\rangle$	Puts a value from memory into a register
Store	$\langle$ **store** r_a, r_v $\rangle$	Puts a value from the register into memory

TABLE I
THIS TABLE CONTAINS A DESCRIPTION FOR ALL THE INSTRUCTIONS IN THE ISA FOR THIS MACHINE

I. INTRODUCTION

We discuss details about various aspects of the processor used in STT, which detail how the store events work.

II. THE INSTRUCTION SET ARCHITECTURE

The STT model uses a simple architecture that only contains five instructions in it, as shown in Table II. The first instruction $\langle$**immed** $\mathsf{r_d}, \mathsf{k}\rangle$, puts an integer value into the register r_d. $\langle$**op** $\mathsf{r_d}, \mathsf{r_a}, \mathsf{r_b}\rangle$ is used to perform an arithmetic operation on the two source registers $\mathsf{r_a}$ and $\mathsf{r_b}$ and assign the value to the destination register r_d. $\langle$**branch** $\mathsf{r_c}, \mathsf{r_d}\rangle$ sets the program counter value to the value in $\mathsf{r_d}$, if $\mathsf{r_c}$ evaluates to true. $\langle$**load** $\mathsf{r_d}, \mathsf{r_a}\rangle$ interprets the value in r_a as a memory address, reads the value at that memory location, and puts it into $\mathsf{r_d}$, the destination register. Lastly, the $\langle$**store** $\mathsf{r_a}, \mathsf{r_v}\rangle$, takes the value in the register $\mathsf{r_v}$ and puts it into the memory address stored in register $\mathsf{r_a}$.

III. THE IN-ORDER PROCESSOR

The in-order processor is a simple model that can run a program consisting of instructions of the above ISA. This processor is able to take a program P as an input, and process the instructions in that program one at a time. The process is shown in Algorithm 1. In every cycle, we first check whether the instruction at the current pc is $\perp$. If so, then the program execution has completed. Otherwise, we need to run an event, corresponding to the instruction at the current pc, and update the state of the in-order processor correspondingly. There are five kinds of events, each one matching the kind of instruction. These events are listed in Table II. These events do similar work as the ISA description for the instructions(see Table II).

IV. OUT-OF-ORDER PROCESSOR

The out-of-order processor is modelled as a state machine, with many of the components found on a real machine, such as ROB, LQ, SQ and others (see Table IV-F , §IV-A). The

Algorithm 1: Inorder processor Logic

```
   /* In order Processor Algorithm              */
   /* Single-Core Processor Logic               */
1  Function Inorder_Processor (P) :
2      $\Sigma \leftarrow \Sigma\_init$
3      $t \leftarrow 0$ ;
4      $halt \leftarrow false$ ;
5      while $\neg halt$ do
6          $(\Sigma, halt) \leftarrow InO\_Logic(P, \Sigma, t)$ ;
7          $t \leftarrow t + 1$ ;
8      end
9  end
   /* Inorder Logic                             */
10 Function InO_Logic ($P, \Sigma, t$) :
11     if $P[\Sigma.pc] = \perp$ then
12         return($\Sigma$, true) ;
13     $e \leftarrow matching\_event(P[\Sigma.pc])$
14     $\Sigma' \leftarrow perform(\Sigma, e, t)$
15     return($\Sigma'$, false)
16 end
```

algorithm for the out-of-order processor is shown below in Algorithm 2. It has three different parts to it, the main parts being the commit logic, fetch logic and execute logic, which update the machine state every cycle.

The Commit Logic (see Algorithm §3) is used by the processor when it is done processing an in-flight instruction. The instruction at the head instruction of the ROB can be committed, and there is an attempt to commit upto COMMIT_WIDTH number of events. In order to perform the commit, several micro-architectural events can possibly be executed, depending on the kind of instruction at the head of the re-order buffer. The algorithm will first generate a commit event, which could be one of five kinds (see Table III), based on the type of instruction. The events each have a pre-condition and a a perform part. The pre-condition sets out certain requirements which should be true for the processor to be able to execute the event. It also calculates various intermediate register values. The perform part lists out the state updates if the event is executed. After generating the commit event, the next step is to check whether the event is enabled. An event is enabled if it is possible to execute both the pre-condition part and the perform part, i.e. the variables in the pre-condition part can be calculated, the conditions required by the pre-condition part are satisfied, and the state updates in

Event	Precondition	Perform
Immediate	$\mathsf{P[pc]} = \langle \textbf{immed}\ \mathsf{r_d, k} \rangle$	$\{\mathsf{pc, mem, reg}\} \to \{\mathsf{pc}+1, \mathsf{mem}, \mathsf{reg}[\mathsf{r_d} \to \mathsf{k}]\}$
Arithmetic	$\mathsf{P[pc]} = \langle \textbf{op}\ \mathsf{r_d, r_a, r_b} \rangle$	$\{\mathsf{pc, mem, reg}\} \to \{\mathsf{pc}+1, \mathsf{mem}, \mathsf{reg}[\mathsf{r_d} \to \mathsf{op}(\mathsf{reg}(\mathsf{r_a}), \mathsf{reg}(\mathsf{r_b}))]\}$
Branch	$\mathsf{P[pc]} = \langle \textbf{branch}\ \mathsf{r_c, r_d} \rangle$ $\mathsf{pc}' = \textbf{if}\ (\mathsf{reg}(\mathsf{r_c}))\ \textbf{then}\ \mathsf{reg}(\mathsf{r_d})\ \textbf{else}\ \mathsf{pc}+1$	$\{\mathsf{pc, mem, reg}\} \to \{\mathsf{pc}', \mathsf{mem}, \mathsf{reg}\}$
Load	$\mathsf{P[pc]} = \langle \textbf{load}\ \mathsf{r_d, r_a} \rangle$	$\{\mathsf{pc, mem, reg}\} \to \{\mathsf{pc}+1, \mathsf{mem}, \mathsf{reg}(\mathsf{r_d} \to \mathsf{mem}(\mathsf{reg}(\mathsf{r_a})))\}$
Store	$\langle \textbf{store}\ \mathsf{r_a, r_v} \rangle$	$\{\mathsf{pc, mem, reg}\} \to \{\mathsf{pc}+1, \mathsf{mem}[\mathsf{reg}(\mathsf{r_a}) \to \mathsf{reg}(\mathsf{r_v})], \mathsf{reg}\}$

TABLE II
TABLE OF MICRO EVENTS AND THEIR EFFECT ON THE MACHINE STATE UPON EXECUTION, FOR AN INORDER PROCESSOR.

the perform part can be done. If the enabled function returns as true, then the machine state is updated using the perform part of the event. We also add this event to the event trace.

The next phase is the Fetch Logic (see §4), in which the processor tries to fetch FETCH_WIDTH new instructions into the core. Whenever a new instruction is fetched, a new fetch event is generated. There are five kinds of fetch events depending on the kind of instruction that is fetched (see §III). They typically update the re-order buffer, with a new instruction. Similar to the commit events, they have a precondition and a post condition, an enabled condition, and a perform function if the enabled condition is satisfied.

The third phase is the Execute Logic (see §5). The main idea here is to perform the key actions associated with that instruction, for example the addition instruction will need to add the contents of two registers and obtain a result. The function loops over the non-committed instructions currently in the re-order buffer, and tries to execute these functions using micro-architectural events, if it is possible at that time. Similar to the previous two functions, it checks to see that the enabled condition for that event is true, and also does a perform for the event. In addition to enabled, the ready (see §IV-A) and !delay functions should also be true for that instruction. The former means that the physical registers which are the inputs to that instruction have been updated to their correct values (see §IV-C) and are ready for use, and the latter means that there is no need to delay the execution of the instruction for security reasons. One reason to delay is if the instruction is a branch instruction, and the event is a BranchFail or BranchSuccess (see III) event in the shadow of a branch, because the branch resolution should not depend on any tainted registers, which could have been manipulated by the attacker(see §IV-E).

Algorithm 2: Single Core STT Processor Model

```
 /* The main loop runs the STT logic
    each cycle and updates the state.
    If the halt condition is true, the
    machine stops execution.          */
1  Function SingleCore(P) :
2      $\sigma \leftarrow \sigma_{\mathsf{init}}$
3      t ← 0
4      halt ← false
5      while ¬halt do
6          $(\sigma, \mathsf{halt}) \leftarrow \mathsf{STT\_Logic}(\mathsf{P}, \sigma, \mathsf{t})$
7          t ← t + 1
8      end
9  end
 /* STT Logic updates the STT state
    each cycle. If there aren't any
    more instructions in the program
    and all instructions in the ROB
    have completed too, the program
    halts.                            */
10 Function STT_Logic(P, $\sigma$, t) :
11     $\sigma_o \leftarrow \sigma$ // State at cycle start
12     Commit_Logic() // Commit instructions
           at head
13     Fetch_Logic() // Fetch and taint
           instructions
14     Execute_Logic() // Execute Instructions

15     $\sigma \leftarrow \mathsf{untaint}(\sigma)$ // Safe instructions
           determined
16     $\mathsf{halt} \leftarrow (\sigma.\mathsf{rob_{head}} = \sigma.\mathsf{rob_{tail}}) \wedge (\mathsf{P[pc]} = \bot)$
17     return $(\sigma, \mathsf{halt})$
18 end
```

A. Components

As listed in Table IV-F, several components are used by the STT out-of-order processor, which we discuss below. The out-of-order machine model is defined as $\sigma \in \mathsf{State_{STT}}$, where $\sigma = (\mathsf{pc, mem, reg, ready, rt, rob, lq, sq, bp, ckpt, C}, \tau, \mathsf{T})$.

1) Program Counter: The program counter, or $\mathsf{pc} \in \mathsf{N}$ is an integer value. It corresponds to the next instruction that needs to be fetched.

2) Memory: The memory used by the system is defined as $\mathsf{mem} : \mathbb{N} \to \mathbb{Z}$. It is a table that maps natural numbers to real numbers, where the natural numbers represent the addresses, and the real numbers represent the value at that address.

3) Register File: The register file is defined as $\mathsf{reg} : \mathsf{PRegID} \to \mathbb{Z}$, which is a function that maps the RegID space to real numbers, which denotes the contents of the registers.

4) Ready Table: The ready table is defined as $\mathsf{ready} : \mathsf{PRegID} \to \mathbb{B}$, which maps a subset of natural numbers, or PRegIDs, to a boolean value, true or false. It consists of the ready state of all the physical registers, i.e. whether their values have been updated for use by another instruction. For example, when an arithmetic instruction first is put into

Algorithm 3: Commit Logic

```
/* Commit Logic */
Function Commit_Logic($\sigma$, t) :
    for i from 1 to COMMIT_WIDTH do
        e ← commit_event($\sigma$.rob[$\sigma$.rob$_{\mathsf{head}}$])
        if enabled($\sigma$, e, t) then
            $\sigma$ ← perform($\sigma$, e, t)
            // $\mathcal{T} = \mathcal{T} +\!\!+ e$
        else
            break;
        end
    end
end
```

Algorithm 4: Fetch Logic

```
/* Fetch Logic */
Function Fetch_Logic(P, $\sigma$, t) :
    for i from 1 to FETCH_WIDTH do
        e ← fetch_event(P[$\sigma$.pc])
        if enabled($\sigma$, e, t) then
            $\sigma$ ← perform($\sigma$, e, t)
            // $\mathcal{T} = \mathcal{T} +\!\!+ e$
            if $e$ = (FetchBranch, $\perp$) then
                break;
            end
    end
end
```

Algorithm 5: Execute Logic

```
/* Execute Logic */
Function Execute_Logic($\sigma$, t) :
    for i from $\sigma$.rob$_{\mathsf{head}}$ to $\sigma$.rob$_{\mathsf{tail}}$ − 1 do
        e ← execute_event($\sigma$, i)
        if ( enabled($\sigma$, e, t) & ready($\sigma_0$, e, t)
        & !delay($\sigma_0$, e, t)) then
            $\sigma$ ← perform($\sigma$, e, t)
            // $\mathcal{T} = \mathcal{T} +\!\!+ e$
            if e = (BranchFail, i) then
                break;
            end
        end
    end
end
```

the ROB during the fetch event, it sets the ready state of its output register to true once it has completed execution, and only executes when its input physical registers have both been set to the ready state too.

5) Renaming Table: The renaming table, known as rt, contains the mapping between the logical register IDs and the physical register IDs, both of which are natural numbers. It is represented as rt : RegId $\rightarrow$ PRegID.

6) Reorder Buffer: The reorder buffer is represented as rob $\in (\mathbb{N} \times \text{DynInstr} \times (\mathbb{B} \cup \perp))^* \times \mathbb{N}$. There are two pointers associated with the ROB entry. The first one is called rob$_{\mathsf{tail}}$, which points to the location after the last fetched instruction, and the other is called rob$_{\mathsf{head}}$, which is the location of the first un-committed entry. Each ROB entry is made up of three entries, namely the pc, which is the program counter, the Θ, which is the dynamic instruction(see IV-C), and b, which is a flag recording the branch prediction if Θ is a branch instruction, and $\perp$ otherwise.

7) Load Queue: The next element is the load queue, or lq $\in (\mathbb{N} \times \mathbb{B} \times (\mathbb{Z} \cup \perp))$. This consists of a set of three values $(\mathsf{i}, \mathsf{f}, \mathsf{t}_{\mathsf{end}})$. i is the index of the corresponding load instruction in the re-order buffer, f is the boolean flag, which is set to true only if the execution has completed for that instruction, and $\mathsf{t}_{\mathsf{end}}$ is the cycle at which the load's execution ends.

8) Store Queue: The store queue sq $\in \mathbb{N}^*$ is a sequence of integers, which are the indices of the store entries in the reorder buffer.

9) Checkpoints: There is also a list of checkpoints, defined as cpkt $\in (\mathbb{N} \times \mathbb{N} \times \mathcal{P}(\text{LRegID} \rightarrow \text{PRegID}))^*$, which is a sequence of tuples (i, pc, rt). i is the index of the ROB instruction that is triggering a checkpoint, pc is the program counter at the position of the checkpoint and the rt is the register renaming table, at the time that the checkpoint was taken.

10) Cache State: The cache state $\mathsf{C} \in \mathsf{N}^*$, which is abstractly represented as a sequence of memory accesses. $\tau \in$ Taint , is information about which part of the state is tainted and which part is not (see §IV-D).

11) Sharing Table: The sharing table $\mathsf{T} \in (\mathbb{N} \times \mathbb{B} \times \mathbb{N} \times \mathbb{B})^*$ is a mapping table that contains address mapping to a natural numbers, which represents the most recent core ID that accessed that addres. These can be used to implement that sharing table based algorithm to determine whether consistency requirements need to be relaxed. If there are multiple processors, then the same sharing table should be shared among all the machines.

B. Microarchitectural Events

Table III has a detailed listing of all the micro-architectural events. Each event hase a pre-condition and an action. In order for the event to be possible to execute, it should be possible to evaluate teh pre-condition, and any requirements of the pre-condition should be satisfied. It should also be possible to execute the actions. We discuss what these events do briefly in this section, and they are also summarized in Table III.

1) Fetch Events: The first set of events are related to the Fetch part of the processor execution. In this part, there are five events mentioned. (FetchImmediate, $\perp$) puts an Immediate type instruction into the ROB, updates the rt table in the renaming step (see §IV-C), and sets its output physical register's state to false in the ready table. The (FetchArithmetic, $\perp$), (FetchBranch, $\perp$) also carry out updates in a similar manner. (FetchBranch, $\perp$) also adds a checkpoint to the checkpoint list, and updates the branch predictor state to a new one. (FetchLoad, $\perp$) event updates the ROB in a similar manner to the other instructions, but it also adds an entry in the load queue. It also adds a new checkpoint in case its validation might fail later, and the processor needs to rexecute the load and all subsequent instructions. The $(\mathsf{FetchStore}, \bot)$ event updates the state using a similar strategy as $\mathsf{FetchImmediate}$.

2) Commit Event: The Commit group of events are executed next. The first one is $(\mathsf{CommitImmediate}, \bot)$. $(\mathsf{CommitArithmetic}, \bot)$ instructions have very similar preconditions, that they should just have that type of instruction at the head of the ROB, and also that the input registers for them are ready, which means that they have completed their execution and updated their output register. and the action they take is to increment the head pointer by one. $(\mathsf{CommitBranch}, \bot)$ is similar too, except that it has no output register, so it does not need to be ready. However, it should not have any checkpoint corresponding to it either, because those are cleared off when the branch resolves.

The $(\mathsf{CommitLoad}, \bot)$ event is next. The pre-condition is that there is a load type instruction at the head of the ROB, it's destination register is ready, the sharing table's state was able to update successfully, and that the validation succeeded too. The actions that are performed by this event are that it increments the head pointer by one and also that it removes its entry from the load queue.

Lastly, the $\mathsf{CommitStore}$ event occurs if a store instruction has an ROB entry at the head. The pre-conditions for the execution are that the two input registers to the store are ready, so that the store knows which memory address to write and also what value to write. Also $\mathsf{updateState}$ should return, indicating that no other core has yet seen this address. If all the pre-conditions are satisfied, then, the memory and cache states are updated. The store queue and the re-order buffer are also updated, by removing the entries corresponding to our instruction, since the instruction has completed.

3) Execute Events: Finally, let us consider the execute events. $\mathsf{ExecuteImmediate}$ instruction has the pre-condition that there should be an $\mathsf{immediate}$ type instruction at the ROB position being considered, and that its output is not yet ready, indicating that it has not yet been executed. The action it takes, if these pre-conditions hold, is that it updates the state of the output register as ready, and also updates the register value in the register file.

The $(\mathsf{ExecuteArithmetic}, \mathsf{i})$ instruction has the pre-conditions that there is a corresponding instruction in the ROB, and the input registers are ready and the output register is not ready. The actions taken are that the register table updates for the destination register, the ready table also updates.

The $(\mathsf{ExecuteBranchSuccess}, \mathsf{i})$ occurs whenever a branch has been deemed to have completed successfully.The pre-condition for this event is that there is an ROB entry for the corresponding branch instruction, the input registers of the branch are ready, and that they exactly match with the predicted branch condition, and also the predicted branch target. A new updated branch predictor state is also calculated. The state updates are that the branch predictor state is updated to the new state and the checkpoint corresponding to the branch is removed from the chekpoint list, because we don't need to revert to it anymore.

The $(\mathsf{ExecuteBranchFailure}, \mathsf{i})$ happens, whenever there is a branch resolves and finds that it has mispredicted. The pre-condition is that there is a branch instruction in the ROB corresponding to this event, the input registers are ready, and either the branch target or branch condition were predicted incorrectly. In the pre-condition, the correct program counter and branch predictor state are also calculated. The state updates carried out are to update the rob, the lq and sq because we need to revert back to a previous state due to branch mispredict. We also update the branch predictor and remove the checkpoint from the checkpoint list, because it is no longer needed.

Let us now consider the load events. The first one is $(\mathsf{ExecuteLoadBeginGets}, \mathsf{i})$ event. The pre-condition for this event is that there is an ROB entry for the event, and its input register $\mathsf{x_a}$ is ready, and the output register $\mathsf{x_d}$ is not ready. This means that it has not yet put a value into $\mathsf{x_d}$. The $\mathsf{storesAreReady}(\mathsf{i})$ value should evaluate to true, because a load can execute out-of-order only if the older stores before it have already determined which address they are storing to. This is so that once the load completes, it will be able to get forwarded values from older stores, if any of them write to the same address that the load is getting its value from. Another parameter calculated is the total latency of the load, which is determined on the basis of the cache state and the address. We set it as a constant, so that the load timing is data oblivious. The $\mathsf{noTaintedInputs}$ function is also used to determine if the load is in the shadow of a branch instruction, in which case it would return true, otherwise false.

The updates to the state include an updated load queue entry to reflect that the load has started, and also an updated cache state, because the cache state may be changed by the load. If the load input was not tainted, then the cache state is updated, otherwise not. This serves a security purpose, because cache state should not be updated by tainted state.

The next event, $(\mathsf{ExecuteLoadEndGets}, \mathsf{i})$. Its pre-condition is that there is a corresponding load instruction in the re-order buffer, and also the time is greater than the end time indicated for that load, meaning that it is sure that the load must have completed, and also, the end time is not $\bot$, meaning it is sure that the load had started. If all these conditions have been met, then we can update the load queue, so that it marks the load as completed, and also fill in the contents of the register $\mathsf{x_d}$ with the value obtained from memory.

The $(\mathsf{ExecuteLoadComplete}, \mathsf{i})$ has the pre-condition that there should be an ROB entry for this event, and that it should also have a load queue entry indicating that the load has successfully completed. The load result will now be calculated, and be updated into the register contents. This is a forwarding check action, which checks the values which may have been written by older stores. If there is any such older store which is writing to the same address, then we update our load result to that value. If there isn't any such older store, then we re-use the same value that we had obtained from memory. The other update to the machine state is to update the ready state of the register $\mathsf{x_d}$ as true, because the value in it can now be made available to younger instructions.

The last event is the $(\mathsf{ExecuteLoadSquash}, \mathsf{i})$ event. This event has a pre-condition, that there should be an event at the head of the ROB corresponding to it, the output register should be ready, meaning the load has completed, it should be possible to update the sharing table and check whether any other core accessed it, and lastly, it should also be possible to check that the validation failed i.e. the value in memory is different from the value seen when the load executed. The state updates carried out, are to revert the state back to when the load was fetched, by reverting the lq, sq, rob and ckpt lists.

C. Renaming step

The renaming step corresponds to the assignment of a new register to the output of the instruction. We need to assign a fresh physical register to the output of that instruction, and also assign the correct physical registers as the input registers, based on dependencies in the program. The renaming step usually is carried out when fetching the instruction, and results are put into an updated rt table, rt'.

The renaming operations are summarized in Table V. New physical registers are assigned for the outputs of the immediate, arithmetic and load instructions, and the renaming table is also updated, mapping the corresponding logical registers to these new physical registers. Second, the dynamic instruction in the ROB uses physical registers instead of the logical registers of the instruction.

D. Taint Tracking

STT [1] uses a taint tracking system in order to determine which instructions are safe, and which ones are unsafe. The taint tracking system that is defined by the STT model carefully selects which registers need to be tainted. One way to do it is so that all the registers, which are not under a branch shadow, will not be tainted, and all other registers will be tainted.

E. Security Argument

Let us define an extended state for the processor, $\kappa = (\sigma, \mathsf{trace}, \mathsf{mispredict}, \mathsf{doomed})$. σ is the out of order machine state. trace is the correct trace of committed pcs for the program, so it is a list of natural numbers. The current trace is the list of pcs of all the instructions in the ROB. mispredict indicates whether the current trace of the processor has diverged from the correct trace. This can be imagined in the following way. Line up the two lists side by side and start tracing them from the start to the finish. At some point, it is possible that the pcs on the two lists do not match up. This is the point of divergence.It could be that at the divergence point, the current list has a different pc than the correct list, or that correct list simply ended, but the current list went on. Lastly, the doomed variable represents the registers which have been allocated by instructions in the incorrect part of the current trace.

κ_1 runs a program according to the Algorithm 2. We also assume that the following property holds for the mispredict flag for κ_1. When the mispredict flag turns 1, then a divergence has occurred compared to the correct trace, and the last matching instruction was a branch instruction. κ_2 is run in lockstep with κ_1, which means that whenever κ_1 tries to execute an event, then κ_2 will try also.

The inductive security property is as follows. Let us assume that κ_1 and κ_2 have the same state at the start of the cycle, other than the contents of their doomed registers. Then, the property of the processor is that

- $\mathcal{T}_1 = \mathcal{T}_2$, which means that the event trace generated during the cycle is exactly the same on both machines.
- At the end of the cycle, the two processors must have the same state, other than the doomed register contents i.e. $\kappa_1 \approx \kappa_2$ (also called low equivalence).

Let us now start the execution to prove that this property holds. First we run the commit events. Whenever the first processor κ_1 runs a commit event, then we also try to run a commit event in the second processor. We observe that in each of the commit events, CommitImmediate, CommitArithmetic, CommitBranch, no register contents are involved in the pre-condition or in the state updates. Therefore, for enabled calculations, if κ_1 is successful in calculating the pre-conditions and updating the state, then κ_2 should also succeed, because it has the same state, other than the doomed registers. Also, if κ_1 fails, then κ_2 should also fail. The changes to the state due to perform should be exactly the same. Next, let us consider the CommitLoad and CommitStore events. For enable pre-condition and calculation and state updates, no doomed registers are involved, because the instruction is at the head of the ROB, and therefore not in a branch shadow. perform does not use any doomed register content to update the state. Therefore, both $\mathsf{kappa_1}$ and $\mathsf{kappa_2}$ should either both execute the event, or both not execute the event, and the final state after the attempt should be the same.

Next, let us consider that all the Fetch events. They don't use any doomed registers in the calculations of perform or enabled. Therefore, both κ_1 and κ_2 should either both execute the event, or both not execute the event, and the final state after the attempt should be the same, other than doomed register contents.

Lastly, let us consider the execute events in the system. For ExecuteImmediate and ExecuteArithmetic, the precondition does not use any doomed register values, and the state update only updates doomed registers with doomed registers. Therefore, it should be possible to show that $\mathsf{kappa_1}$ and $\mathsf{kappa_2}$ should either both execute the event, or both not execute the event, and the final state after the attempt should be the same.

ExecuteBranchSuccess and ExecuteBranchFail, we note that the !delay condition in Algorithm 5, which will delay resolution of any branches in a branch shadow. If delayed in κ_1, it should also be delayed in κ_2, because if there is an older unresolved branch in κ_1, it should also be there in κ_2.

Similarly, if it is not delayed in κ_1, then it should also not be delayed in κ_2, because if there isn't an older unresolved branch in κ_1 there shouldn't be one in κ_2 either. All this because κ_1 and κ_2 have same state other than doomed register contents. In the latter case, since the pre-condition and state updates don't depend on the doomed registers, both κ_1 and κ_2 should either execute the event or not execute the event, and have the same state updates in case they both executed it.

Let us consider the load events. ExecuteLoadBeginGets input registers may or may not be doomed. If it is not in a branch shadow, it is definitely not doomed. In the pre-condition, all the calculations do not depend on the value of the registers. In state updates, if there was a doomed input, then the update function should be able to determine that it was in the branch shadow, and not update the cache state based on the register contents, if it was. Other state updates do not depend on register updates. Therefore, it should be possible to show that κ_1 and κ_2 should either both execute the event, or both not execute the event, and the final state after the attempt should be the same, other than doomed register contents.

For ExecuteLoadEndGets, no changes based on register contents in the pre-condition or the state updates so it should be possible to show that, κ_1 and kappa_2 should either both execute the event, or both not execute the event, and the final state after the attempt should be the same.

For ExecuteLoadComplete, the input register may or may not be doomed. If the instruction is not in the branch shadow it is not doomed. If it is in the branch shadow, then it may be doomed. In the pre-condition, if κ_1 is able to make all the calculations, it should be possible to show that κ_2 also can make them, and vice versa. For the first two calculations, this is trivially true, because it does not depend on doomed register content. For the third calculation, it is a memory access, to a location that might be dependent on doomed registers. However, this should always be possible. The state updates cause an update to a doomed register content (if the instruction is doomed), and also the ready table. If κ_1 can make these updates, then it should be possible to show that κ_2 can as well, and vice versa. The final state should still be the same other than doomed register contents, because the ready table update does not depend on doomed register values, and the register file updates a doomed register value, so it does not matter what it updates the value to, on the two machines.

F. Future Work

We have made a minimal set of changes to the existing STT model in order to make it more compatible with an SRCP optimization. The main change was in the CommitLoad function, where we added in a validation and a sharing state table update. It should be possible to also add in validation squashes, when the validation fails, and put together multiple copies of this processor for a multicore model, with minimal changes.

References

[1] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 954–968.

Event	Precondition	Perform
$(\mathsf{FetchImmediate}, \perp)$	$\mathsf{P}[\mathsf{pc}] = \langle \textbf{immed}\ \mathsf{r_d}, \mathsf{k} \rangle$ $(\mathsf{rt}', \Theta) = \mathsf{rename}(\sigma, \theta)$	$\mathsf{pc} \leftarrow \mathsf{pc} + 1; \mathsf{rt} \leftarrow \mathsf{rt}';$ $\mathsf{rob} \leftarrow [\mathsf{rob} +\!\!+ [(\mathsf{pc}, \Theta, \perp)]; \mathsf{ready}[\mathsf{x_d} \rightarrow \mathsf{false}]$
$(\mathsf{FetchArithmetic}, \perp)$	$\mathsf{P}[\mathsf{pc}] = \langle \textbf{op}\ \mathsf{r_d}, \mathsf{r_a}, \mathsf{r_b} \rangle$ $(\mathsf{rt}', \Theta) = \mathsf{rename}(\sigma, \theta)$	$\mathsf{rob} \leftarrow [\mathsf{rob} +\!\!+ [(\mathsf{pc}, \Theta, \perp)];$ $\mathsf{ready}[\mathsf{x_d} \rightarrow \mathsf{false}]; \mathsf{rt} \leftarrow \mathsf{rt}'; \mathsf{pc} \leftarrow \mathsf{pc} + 1$ $\mathsf{lq} \leftarrow \mathsf{lq} +\!\!+ (\mathsf{i}, \mathsf{false}, \perp)$
$(\mathsf{FetchBranch}, \perp)$	$\mathsf{P}[\mathsf{pc}] = \langle \textbf{branch}\ \mathsf{r_c}, \mathsf{r_d} \rangle$ $(\mathsf{rt}', \Theta) = \mathsf{rename}(\sigma, \theta)$ $\mathsf{newCkpt} = (\mathsf{rob_{tail}}, \mathsf{pc}, \mathsf{rt})$ $\mathsf{bp.predict}(\mathsf{pc}) = (\mathsf{spc}, \mathsf{b});$ $\mathsf{bp}' = \mathsf{bp.update}(\mathsf{b});$ $\mathsf{pc}' = (\textbf{if}\ \mathsf{b}\ \textbf{then}\ \mathsf{spc}\ \textbf{else}\ \mathsf{pc} + 1)$	$\mathsf{rob} \leftarrow [\mathsf{rob} +\!\!+ [(\mathsf{pc}, \Theta, \perp)]\ ;$ $\mathsf{ready}[\mathsf{x_d} \rightarrow \mathsf{false}]; \mathsf{rt} \leftarrow \mathsf{rt}'; \mathsf{pc} \leftarrow \mathsf{pc} + 1\ ;$ $\mathsf{bp} \leftarrow \mathsf{bp}'; \mathsf{ckpt} \leftarrow [\mathsf{ckpt} +\!\!+ \mathsf{newCkpt}]$
$(\mathsf{FetchLoad}, \perp)$	$\mathsf{P}[\mathsf{pc}] = \langle \textbf{load}\ \mathsf{r_d}, \mathsf{r_a} \rangle$ $(\mathsf{rt}', \Theta) = \mathsf{rename}(\sigma, \theta)$ $\mathsf{x_d} = \mathsf{rt}'(\mathsf{r_d})$ $\mathsf{newCpkt} = (\mathsf{rob_{tail}}, \mathsf{pc}, \mathsf{rt})$	$\mathsf{rob} \leftarrow [\mathsf{rob} +\!\!+ [(\mathsf{pc}, \Theta, \perp)]$ $\mathsf{ready}[\mathrm{x\_d} \rightarrow \mathsf{false}]; \mathsf{rt} \leftarrow \mathsf{rt}'$ $\mathsf{pc} \leftarrow \mathsf{pc} + 1$ $\mathsf{ckpt} \leftarrow \mathsf{ckpt} +\!\!+ \mathsf{newCkpt}$
$(\mathsf{FetchStore}, \perp)$	$\mathsf{P}[\mathsf{pc}] = \langle \textbf{store}\ \mathsf{r_a}, \mathsf{r_v} \rangle$ $(\mathsf{rt}', \Theta) = \mathsf{rename}(\sigma, \theta)$	$\mathsf{rob} \leftarrow [\mathsf{rob} +\!\!+ [(\mathsf{pc}, \Theta, \perp)]\ ;$ $\mathsf{ready}[\mathsf{x_d} \rightarrow \mathsf{false}]; \mathsf{rt} \leftarrow \mathsf{rt}'; \mathsf{pc} \leftarrow \mathsf{pc} + 1$
$(\mathsf{CommitImmediate}, \perp)$	$\mathsf{rob_{seq}}[\mathsf{rob_{head}}] = \langle \_; \textbf{immed}\ \mathsf{x_d}; \_ \rangle$ $\mathsf{ready}(\mathsf{x_d}) = \mathsf{true}$	$(\mathsf{rob_{seq}}, \mathsf{rob_{head}}) \leftarrow (\mathsf{rob_{seq}}, \mathsf{rob_{head}} + 1)$
$(\mathsf{CommitArithmetic}, \perp)$	$\mathsf{rob_{seq}}[\mathsf{rob_{head}}] = \langle \_; \textbf{op}\ \mathsf{x_d}, \mathsf{x_a}, \mathsf{x_b}; \_ \rangle$ $\mathsf{ready}(\mathsf{x_d}) = \mathsf{true}$	$(\mathsf{rob_{seq}}, \mathsf{rob_{head}}) \leftarrow (\mathsf{rob_{seq}}, \mathsf{rob_{head}} + 1)$
$(\mathsf{CommitBranch}, \perp)$	$\mathsf{rob_{seq}}[\mathsf{rob_{head}}] = \langle \_; \textbf{branch}\ \mathsf{x_c}, \mathsf{x_d}; \_ \rangle$ $(\mathsf{rob_{head}}, \mathsf{pc}', \mathsf{rt}') \notin \mathsf{ckpt}$	$(\mathsf{rob_{seq}}, \mathsf{rob_{head}}) \leftarrow (\mathsf{rob_{seq}}, \mathsf{rob_{head}} + 1)$
$(\mathsf{CommitLoad}, \perp)$	$\mathsf{rob_{seq}}[\mathsf{rob_{head}}] = \langle \_; \textbf{load}\ \mathsf{x_d}, \mathsf{x_a}; \_ \rangle$ $\mathsf{ready}(\mathsf{x_d}) = \mathsf{true}$ $\mathsf{s} = \mathsf{updateState}(\mathsf{T}, \mathsf{reg}(\mathsf{x_a}))$ $\mathsf{validate}(\mathsf{s}, \mathsf{reg}(\mathsf{x_d}), \mathsf{mem}[\mathsf{reg}(\mathsf{x_a})])$	$(\mathsf{rob_{seq}}, \mathsf{rob_{head}}) \leftarrow (\mathsf{rob_{seq}}, \mathsf{rob_{head}} + 1)$ $\mathsf{lq} \leftarrow \mathsf{lq}\vert_{\neq \mathsf{rob_{head}}}$
$(\mathsf{CommitStore}, \perp)$	$\mathsf{rob_{seq}}[\mathsf{rob_{head}}] = \langle \_; \textbf{store}\ \mathsf{x_a}, \mathsf{x_v}; \_ \rangle$ $\mathsf{ready}(\mathsf{x_a}) = \mathsf{true}; \mathsf{ready}(\mathsf{x_v}) = \mathsf{true}$ $\mathsf{updateState}(\mathsf{T}, \mathsf{reg}(\mathsf{x_a}))$	$(\mathsf{rob_{seq}}, \mathsf{rob_{head}}) \leftarrow (\mathsf{rob_{seq}}, \mathsf{rob_{head}} + 1)$ $\mathsf{sq} \leftarrow \mathsf{sq}\vert_{\neq \mathsf{rob_{head}}}, \mathsf{C} \leftarrow \mathsf{C} +\!\!+ \mathsf{reg}(\mathsf{x_a})$
$(\mathsf{ExecuteImmediate}, \mathsf{i})$	$\mathsf{rob}[\mathsf{i}] = (\_; \langle \textbf{immed}\ \mathsf{x_d}, \mathsf{k} \rangle; \_)$ $\neg \mathsf{ready}(\mathsf{x_d})$	$\mathsf{reg}[\mathsf{x_d} \rightarrow \mathsf{k}]\ ; \mathsf{ready}[\mathsf{x_d} \rightarrow \mathsf{true}]$
$(\mathsf{ExecuteArithmetic}, \mathsf{i})$	$\mathsf{rob}[\mathsf{i}] = (\_, \langle \textbf{op}\ \mathsf{x_d}, \mathsf{x_a}, \mathsf{x_b} \rangle, \_)$ $\neg \mathsf{ready}(\mathsf{x_d}); \mathsf{ready}(\mathsf{x_a})\ ; \mathsf{ready}(\mathsf{x_b})$	$\mathsf{reg}[\mathsf{x_d} \rightarrow \mathsf{op}(\mathsf{reg}(\mathsf{x_a}), \mathsf{reg}(\mathsf{x_b}))]\ ;$ $\mathsf{ready}[\mathsf{x_d} \rightarrow \mathsf{true}]$
$(\mathsf{ExecuteBranchSuccess}, \mathsf{i})$	$\mathsf{rob}[\mathsf{i}] = (\mathsf{pc}'; \langle \textbf{branch}\ \mathsf{x_d}, \mathsf{x_c} \rangle; (\mathsf{spc}, \mathsf{b}))$ $\mathsf{ready}(\mathsf{x_d}); \mathsf{ready}(\mathsf{x_c}); \mathsf{b} = \mathsf{reg}(\mathsf{x_c})$ $\mathsf{spc} = \mathsf{reg}(\mathsf{x_d})\ ;$ $\mathsf{bp}' = \mathsf{bp.update}(\mathsf{pc}', \mathsf{reg}(\mathsf{x_c}), \mathsf{reg}(\mathsf{x_d}))$	$\mathsf{bp} \leftarrow \mathsf{bp}'; \mathsf{ckpt} \leftarrow \mathsf{ckpt}\vert_{\neq \mathsf{i}}$
$(\mathsf{ExecuteBranchFail}, \mathsf{i})$	$\mathsf{rob}[\mathsf{i}] = (\mathsf{pc}'; \langle \textbf{branch}\ \mathsf{x_d}, \mathsf{x_c} \rangle; (\mathsf{spc}, \mathsf{b}))$ $\mathsf{ready}(\mathsf{x_d}); \mathsf{ready}(\mathsf{x_c})\ ;$ $\mathsf{b} \neq \mathsf{reg}(\mathsf{x_c}) \vee \mathsf{spc} \neq \mathsf{reg}(\mathsf{x_d});$ $(\mathsf{i}, \mathsf{pc}', \mathsf{rt}') \in \mathsf{ckpt}\ ;$ $\mathsf{pc}'' = (\textbf{if}\ \mathsf{reg}(\mathsf{x_c})\textbf{then}\ \mathsf{reg}(\mathsf{x_d})\textbf{else}\ \mathsf{pc}' + 1);$ $\mathsf{bp}' = \mathsf{bp.update}(\mathsf{pc}', \mathsf{reg}(\mathsf{x_c}), \mathsf{reg}(\mathsf{x_d}))$	$\mathsf{rob} \leftarrow \mathsf{rob}\vert_{\mathsf{i}}; \mathsf{lq} \leftarrow \mathsf{lq}\vert_{<\mathsf{i}}; \mathsf{sq} \leftarrow \mathsf{sq}\vert_{<\mathsf{i}}\ ;$ $\mathsf{bp} \leftarrow \mathsf{bp}'; \mathsf{ckpt} \leftarrow \mathsf{ckpt}\vert_{<\mathsf{i}}$
$(\mathsf{ExecuteLoadBeginGets}, \mathsf{i})$	$\mathsf{rob}[\mathsf{i}] = (\_; \langle \textbf{load}\ \mathsf{x_d}, \mathsf{x_a} \rangle; \_); \mathsf{ready}(\mathsf{x_a})$ $\neg \mathsf{ready}(\mathsf{x_d}); \mathsf{storesAreReady}(\mathsf{i})\ ;$ $\mathsf{x} = \mathsf{noTaintedInputs}(\sigma, \mathsf{i})\ ;$ $\mathsf{t_{end}} = \mathsf{t} + \mathsf{Constant}$	$\mathsf{lq} +\!\!+ (\mathsf{i}, \mathsf{false}, \perp) +\!\!+ \mathsf{lq}' \leftarrow \mathsf{lq} +\!\!+ (\mathsf{i}, \mathsf{false}, \mathsf{t_{end}}) +\!\!+ \mathsf{lq}'\ ;$ $\mathsf{C} \leftarrow \mathsf{update}(\mathsf{C}, \mathsf{reg}(\mathsf{x_a}), \mathsf{x})$
$(\mathsf{ExecuteLoadEndGets}, \mathsf{i})$	$\mathsf{rob}[\mathsf{i}] = (\_; \langle \textbf{load}\ \mathsf{x_d}, \mathsf{x_a} \rangle; \_)\ ;$ $\mathsf{t} \geq \mathsf{t_{end}}; \mathsf{t_{end}} \neq \perp$	$\mathsf{reg}[\mathsf{x_d} \rightarrow \mathsf{mem}(\mathsf{reg}(\mathsf{x_a}))]\ ;$ $\mathsf{lq} +\!\!+ (\mathsf{i}, \mathsf{false}, \mathsf{t_{end}}) +\!\!+ \mathsf{lq}' \leftarrow \mathsf{lq} +\!\!+ (\mathsf{i}, \mathsf{true}, \mathsf{t_{end}}) +\!\!+ \mathsf{lq}'$
$(\mathsf{ExecuteLoadComplete}, \mathsf{i})$	$\mathsf{rob}[\mathsf{i}] = (\_; \langle \textbf{load}\ \mathsf{x_d}, \mathsf{x_a} \rangle; \_)\ ;$ $(\mathsf{i}, \mathsf{true}, \_) \in \mathsf{lq}$ $\mathsf{result} = \mathsf{loadResult}(\mathsf{i}, \mathsf{x_a}, \mathsf{reg}(\mathsf{x_d}))$	$\mathsf{reg}[\mathsf{x_d} \rightarrow \mathsf{res}]; \mathsf{ready}[\mathsf{x_d} \rightarrow \mathsf{true}]$

TABLE III
TABLE OF MICRO EVENTS AND THEIR EFFECT ON THE MACHINE STATE UPON EXECUTION.

Component	Symbol	Description
Reorder Buffer	rob	A list of tuples. Head and tail of the list are marked as $\mathsf{rob}_{\mathsf{head}}$ and $\mathsf{rob}_{\mathsf{tail}}$.
Program Counter	pc	The current program counter.
Branch Checkpoint	ckpt	Information to restore machine state after branch fail
Sharing Table	T	Table storing sharer state of memory addresses. It maps an address to an allocator core, and a state, either E/S/I.
Cache	C	Cache state, represented abstractly as a list of memory accesses
Taint metadata	τ	Table containing information about the tainted registers
Branch Predictor	bp	Branch predictor
Ready Table	ready	Specifies which registers are ready for use.
Register File	reg	Register content table
Program Counter	pc	Current program counter
Remapping Table	rt	Maps logical register identifier, to a physical register identifier
Load Queue	lq	Tuples containing information per load in the ROB
Store Queue	sq	List of tuples corresponding to the stores in the ROB

TABLE IV
OUT OF ORDER MACHINE STATE COMPONENTS

Instruction	New Register	Renaming Action
$\theta = \langle \textbf{immed}\ \mathsf{r_d}, \mathsf{k}\rangle$	$\mathsf{x_d} = \mathsf{fresh}(\sigma)$	$\sigma.\mathsf{rt}(\mathsf{r_d} \mapsto \mathsf{x_d})$
$\theta = \langle \textbf{op}\ \mathsf{r_d}, \mathsf{r_a}, \mathsf{r_b}\rangle$	$\mathsf{x_d} = \mathsf{fresh}(\sigma)$	$\sigma.\mathsf{rt}(\mathsf{r_d} \mapsto \mathsf{x_d}), \langle \textbf{op}\ \mathsf{x_d}, \sigma.\mathsf{rt}(\mathsf{r_a}), \sigma.\mathsf{rt}(\mathsf{r_b}))\rangle$
$\theta = \langle \textbf{branch}\ \mathsf{r_d}, \mathsf{r_a}, \mathsf{r_b}\rangle$	NA	$\sigma.\mathsf{rt}, \langle \textbf{branch}\ \mathsf{x_d}, \sigma.\mathsf{rt}(\mathsf{r_c}), \sigma.\mathsf{rt}(\mathsf{r_d}))\rangle$
$\theta = \langle \textbf{load}\ \mathsf{r_d}, \mathsf{r_a}\rangle$	$\mathsf{x_d} = \mathsf{fresh}(\sigma)$	$\sigma.\mathsf{rt}(\mathsf{r_d} \mapsto \mathsf{x_d}), \langle \textbf{load}\ \mathsf{x_d}, \sigma.\mathsf{rt}(\mathsf{r_a}))\rangle$
$\theta = \langle \textbf{store}\ \mathsf{r_a}, \mathsf{r_v}\rangle$	NA	$\sigma.\mathsf{rt}, \langle \textbf{store}\ \sigma.\mathsf{rt}(\mathsf{r_a}), \sigma.\mathsf{rt}(\mathsf{r_v})\rangle$

TABLE V
THIS TABLE SHOWS THE RENAMING SCHEME.