Introduction

Motivation for Proposing a new Microarchitecture
Background and Literature Study

Resource-Flow execution model

One of our incentives for the design of a new microarchitecture was to exploit as much speculation as possible. We believe that achieving high IPC requires us to go beyond the traditional data and control flow dependency constraints in executing a program and allow for more parallel execution of the code. In the current microarchitectures that perform speculative execution, access to the reorder buffer becomes very problematic as the number of instructions being speculatively executed concurrently grows [D19]. This is mainly due to the contention for the centralized resources such as data registers within the reorder buffer. In the proposed resource flow execution model the instructions with the highest priority that are not yet executed are assigned to free execution resources. The assignment is done regardless of whether the input operands of the instructions have the correct value (data flow constraints) and regardless of the necessity of the execution of the instructions (control flow constraints). This means that the execution is guided based on the availability of resources. The rest of the execution time is spent re-executing the instructions that have had an incorrect speculative input or operand, so as to end up with a programmatically correct execution of the code. As we will show in the rest of this section, in the Resource-Flow execution model there is no need for explicit renaming of registers or any reorder buffers which would eliminate, the contention for centralized resources. This model provides us with methods to executes standard *control flow* based programs that go beyond the standard control flow model, the *data flow* model[U1] and the Superspeculative model[U10].

Time Tags, Forwarding and Snooping

In the Resource-Flow execution model, instructions can be executed speculatively and in an out of order fashion. We use time tags to enforce the correct data and control dependencies among the instructions to realize speculative data-flow execution of code. A time tag is a small integer that uniquely identifies the position of an instruction or an operand in program ordered time with respect to the most recently retired instructions. Time tags are used in common processors to squash instruction results occurring after a mispredicted branch, as well as to maintain instruction order in general. Time tags as used in our microarchitecture were originally proposed in the Warp Engine [U2]. This machine relied on the use of floating point numbers for the time tags In our microarchitecture, time tag values refer to the issue slots in the execution window and are small integer values.

To assign the time tags, the oldest issued instruction in flight that is neither yet committed nor squashed would usually have a time tag value of zero. More recently dispatched instructions take on increasingly larger values the further ahead they are in the program ordered instruction stream. As a group of instructions with the lowest value time tags are retired, all of the time tags for all instructions and operands are decremented by the number of instructions just retired. This will keep the next instruction to be retired in having the time tag value of zero and therefore prevents time tags from growing indefinitely.

Both renaming and time-tagging assume forward broadcast of instruction result information on a bus, snooped by all later instructions. In the resource flow model, the forward broadcast of the identifying address of the output operand and its value is accompanied with the instruction time tag and path ID (for those microarchitectures using multipath execution) of the forwarded result. The time tag and path ID will be used by subsequent instructions in the execution window to determine if the operand has the desired address value and therefore should be snarfed as an input that will trigger its execution or re-execution.

The issue slots in our microarchitecture are modified to handle instruction execution and forwarding and snooping of instruction results. We call this modified issue slot an *Active Station* (AS). Each active station has a Last Snarfed Time Tag (LSTT) register. When an instruction executes, it broadcasts its time tag (ResTT). Snooping stations compare the broadcast result time tag (ResTT) with that held in LSTT. If the result is later than last snarfed (LSTT $\leq$ ResTT), or the LSTT has not been loaded yet, and the register address match then the result value is snarfed. The snarfing instruction is executed and the LSTT is loaded with ResTT. This ensures that only the most-recent previous version of an operand is used by an instruction. Further description of the active station internal structure is provided in section %%.

The information associated with each operand that is broadcasted from one instruction to subsequent ones is referred to as a *transaction*, and generally consists of:

- Transaction type
- Operand Type
- Path ID
- Time tag of the originating instruction
- Identifying address
- Data value for this operand

True flow dependencies are enforced through continuous snooping of theses transactions by each dispatched instruction residing in an issue slot that receives the transaction.

Figure 2 is an example of how time tags are used to enforce correct execution of instructions by enforcing data dependencies. Part (a) shows a fragment of a MIPS code. For simplicity, in this example we assume that there are no branches among these instructions and at the end all of them will be committed. Each instruction is assigned to an active station with a unique time tag.

inst.. Label	Time Tag	Inst. Mnemonic	
I1	1	lui	r3,0x8002
…			
I3	3	lw	r8,-29(r28)
I4	4	addiu	r5,r8,16
I5	5	lw	r3,-26(r28)
…			
I7	7	addiu	r5,r1,32
I8	8	xor	r8,r3,r5
I9	9	sw	r8,-32(r28)

(a) Code fragment

Cycle	Execute	Forward	Snarf
-1		Ix(r28),Iy(r1)	I3(r28),I5(r28),I7(r1),I9(r28)
+0	I1,I3,I5,I7		
+1		I1(r3),I7(r5)	I8(r3,r5)
+2	I8	I3(r8),I5(r3)	I4(r8),I8(r3),I9(r8)
+3	I4,I8,I9		
+4		I4(r5),I8(r8)	I9(r8)
+5	I9		

(b) Execution schedule

Figure 2. Time tagged execution of code sample

Figure 2(b) illustrates the steps required to execute this code fragment. For simplicity, we assume that each instruction can be executed independent of the other one and there is no limit on the number of transactions that can be forwarded at each cycle. The first column shows the relative clock cycles. The next three columns list the stations that are either executing an instruction, forwarding a value or snarfing a snooped value. The notation that is used is $I_x(r_y)$ where I_x is the instruction label and r_y is the register that is either forwarded or snarfed in. Note that snarfing is done in parallel with execution or forwarding.

At clock 0 instruction I1, I3, I5 and I7 execute in parallel. The execution is a result of the snarfing of r1 and r28 registers in the previous cycle. Assuming two cycles for the execution of load and store instructions and one cycle for the rest of the instructions in this example, instruction I1 and I7 will have their results ready in the next clock. The new value for register r3 and r5 is forwarded in clock 1 and is snarfed at I8. In the next clock, instruction I8 will execute using its newly read register value. Normally I8 should forward the new result, but since I5 is sending out a new value for r3, the I8 unit snarfs the new value for r3. This results in re-execution of instruction at I8 which happens at clock +3. In the next cycle, I4 sends a new value for register r5 with a time tag of 4. But since the last value of r5 received by I8 had a time tag of 7, which is greater than 4, the new value is ignored by I8 and do not result in another re-execution.

Relay forwarding?

Memory operations, Nullifying and Backward transactions

With the increasing size of the execution window, there is a higher probability that the memory values generated by store operations will be used by other load operations in the window. Our simulations show that for an execution window with 512 issue slots, an average of %xx of the memory operations can be satisfied in the window without having to go to the higher levels of memory hierarchy. This means that if we could somehow satisfy a portion of the memory operations in the execution window, it will result in less pressure on the higher levels of memory hierarchy and an improved overall performance.

To enforce the true memory dependencies, we could still use the forwarding and snooping mechanism introduced in the last section. There are, however, limitations to the applicability of this strategy. The strategy depends upon the fact that the addresses of the architected register operands are fixed. This property is not true for memory operands. The memory address is generally computed using a fixed displacement and a register value. Due to the speculative nature of this microarchitecture, the register value for the memory operation could be incorrect, resulting in an incorrect memory address.

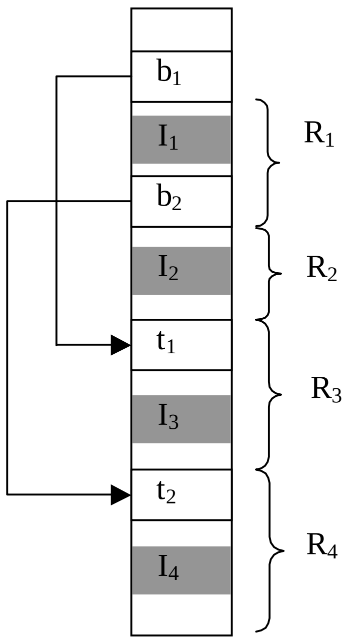
A load operation that snoops for an incorrect memory address will miss the opportunity to snarf the correct memory value forwarded by a previous store operation. It is therefore necessary for a load operation to initiate a memory request using its resolved memory address. In any other architecture, the request normally goes to the first level D-cache. In our architecture, there is a good chance that the memory value is still in the execution window and has not yet written back into the D-cache. Based on this observation, we decided to send the memory requests to both D-cache and earlier in program order issue slots. If there is any store operation with the same address in the previous issue slots, it will forward the memory value, which will be subsequently used by the load instruction. Otherwise the memory value received form the D-cache is used. The requests to the D-cache are sent using a set of shared horizontal buses. The requests to the previous issue slots are handled using a set of *backwarding buses.* These buses are essentially similar to the forwarding buses except that they are mainly used to send data requests to the instructions earlier in the program order. More discussion on memory structure and busing will be presented in later sections.

Another difficulty with using the forwarding and snooping strategy for memory operation is that a value forwarded by an store operation using a wrong memory address could be incorrectly snarfed by another load operation with the same address. This presents a problem for the correct enforcement of memory operand dependencies. We solved this problem by forcing the store operation to notify subsequent load instructions to ignore its previously forwarded value. To do so, we define *nullify transaction* as a new type of transaction which has the property of canceling the effect of a previous store transaction. Any store instruction that has previously forwarded a memory value with the incorrect speculative address, will send a nullify transaction upon re-execution. This transaction is seen by all the subsequent load instructions. Any load instruction that has snarfed a value sent from that specific store instruction will ignore that value and sends anew request to the memory and on the backwarding buses.

Hardware Predication

Predicated execution is shown to be an effective approach for handling conditional branches []. In our microarchitecture, the predicates are generated at run-time. Each instruction computes its own enabling predicate by snooping for and snarfing predicate operands that are forwarded to it by earlier instructions from the program-ordered past. They are evaluated solely with hardware, allowing the use of legacy code. Full hardware based predication is a new implementation of Minimal Control Dependencies. With

MCD, all branches may execute concurrently, and the instructions after a branch domain [U15] may execute independently of the branch.



Branch Domain variables
$D_{b1}(I_1) = D_{b1}(I_2) = D_{b2}(I_2) = D_{b2}(I_3) = 1$
$D_{b1}(I_3) = D_{b2}(I_4) = 0$
Assuming b_1 is enabled and predicted not-taken:
$CEP(I_1) = b_1$
$CEP(I_2) = CEP(I_3) = CEP(I_4) = b_2$
Assuming b_1 is predicted taken, b_2 is disabled:
$CEP(I_1) = CEP(I_2) = CEP(I_3) = CEP(I_4) = b_1$

Figure 3. Branch Domain and Closest Enabled Previous Branch

In this section we propose a hardware predication mechanism that can be implemented using our resource flow methodology. Figure xx shows a program code sequence with two branches b1 and b2. with corresponding targets t1 and t2. The two branches divide the code into 4 regions as is shown in the figure. The execution of the instructions in regions R1, R2 and R3 depends on the outcome of the b1 and b2 whereas the instructions in R4 are executed independent of the branch outcomes. In this section, we are proposing a new scheme for assigning an *enabling predicate* to every instruction in each region based on the outcome of branches. If the enabling predicate value is one, then the instruction will be executed; otherwise it is disabled. The following definitions are used in the description of our scheme.

T_b: a binary value, set to one if the branch *b* is predicted taken
ex_j: a binary value assigned to each instruction I_j and specifies whether the instruction is executed or otherwise disabled.
$D_b(I_j)$: a binary value, set to one if the instructions *j* is in the domain of branch *b*.
$CEP(I_j)$: Closest Enabled Previous branch to instruction I_j in the static program-order.

Figure 3 shows an example along the value of some the above defined variables. The $D_b(I)$ function is independent of the outcome of the branch as only depends on the static order of instructions in the code. *CEP(I)* on the other hand, is a function of the outcome of other branches and will change during the course of speculative execution of code.

Using the above definitions, we can see that

$$ex_j^* = T_b \,.\, D_b(I_j)$$

where $b = CEP(I_j)$.

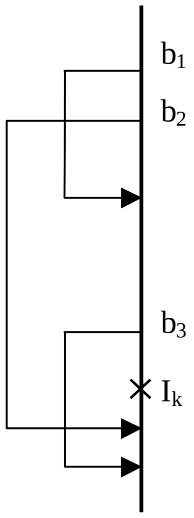
This equation simply tells us that if an instruction I_j is in the domain of an enabled branch, and the branch is taken, it will not be executed. If the branch is out of the domain, then its execution is independent of the outcome of the branch.

Equation (1) is the base for our dynamic predication scheme. To set the predication for each instruction $I_{j,}$, it is sufficient to find the $CEP(I_j)$ branch and its outcome.
This is a simple task in our scheme. Each instruction slot has a pair of CEP_TT registers to keep the time tag of its current CEP branch and corresponding target. If an earlier branch is enabled, a *Predicate* transaction is sent on the forwarding bus with the branch Time tag, its target address and whether it is a taken branch or not. Each subsequent instruction that snoops this transaction checks to see if the time tag of the newly enabled branch is greater than its current CEP_TT value. If so, the new branch is closer to this instruction and therefore it will replace the older CEP branch.

If a branch such as *b* is disabled, the later instructions need to obtain the new CEP branch. The new branch is simply *CEP(b)* because there is no other enabled branch between it and *b*. In our scheme this is handled by forwarding an *invalidate* transaction by branch *b*. An invalidate transaction contains the time tag of *b*, the time tag of *CEP(b)* and the branch domain information of *CEP(b)*. This information was stored at *b* when *CEP(b)* forwarded its predicate transaction.

To find the domain of a CEP branch, a simple approach is to just compare the branch target address with the instruction address. It is a however better to use time tags for this purpose. To do so, a branch needs to forward the time tag of its target instruction instead of its target address. The branch target time tag can be easily calculated by adding the branch displacement to the branch time tag value.

Another improvement to the above scheme can be made by noticing that a not-taken enabled branch does not need to send out any transaction on the forwarding bus. This is due to the fact that a not taken branch essentially acts like a NOP operation. The advantage is a reduction of the number of transactions on the bus.



(a)

	Branch Status			Inst. Status		Transactions	
Cycle	b_1	b_2	b_3	I_k	CEP	Pred.	Inval.
0	T	D	N	E	b_3		
+1	N	D	N	E	b_3	b_1.nt	
+2	N	T	N	E	b_3	b_2.t	
+3	N	T	D	E	b_3		$b_3(b_2)$
+4	N	T	D	D	b_2		
+10	T	T	D	D	b_2	b_1.t	
+11	T	D	D	D	b_2		$b_2(b_1)$
+12	T	D	N	D	b_2	b_3.nt	
+13	T	D	N	E	b_3		

(b)

Figure 4 shows an example of how our dynamic predicate work. The first column in the table lists the relative clock cycles. The next three columns lists the branch status. The "D", "T' and "NT" entries correspond to disabled, taken and not-taken status for each branch. The next two column list the I_k Instruction status. "E" and "D" stand for enabled and disabled respectively. The CEP column shows the *CEP*(I_k) branch at any cycle. The next two columns show the transactions on the bus. The "pred." column lists the predicate forward transactions for each branch along with its status. The "inval." column lists the invalidating transactions. the branch name in the parenthesis corresponds to the new CEP forwarded by the invalidated branch.

In the example, it is assumed that b_1 and b_3 are initially predicted taken and not-taken respectively and a as a result I_k is enabled. In the next clock, b_1 changes its direction to a not-taken branch and sends a forwarding transaction on the bus. This transaction will enable b_2 which is also predicted taken. b2 will send a predication transaction on the forwarding bus which is snooped by b3. As a result, b3 will be disabled in clock 3 and will send an invalidating transaction with b2 as its CEP branch. instruction I_k will see this transaction and switch its CEP to be b2 and as a results will be disabled.
The bottom section of the table in figure %% shows what will happen if b_1 changes back to a taken status. A new set of transactions will follow that eventually enable I_k.

As can be seen from the example, the cost of hardware predication is low, since most of the extra state storage only takes a few bits in the issue slot. This hardware stays same for all issue slots and columns