Appears in the 11^{th} International Conference on Architectural Support for Programming Languages and Operating Systems

Scalable Selective Re-Execution for EDGE Architectures

Rajagopalan Desikan† Simha Sethumadhavan Doug Burger
Stephen W. Keckler

Computer Architecture and Technology Laboratory
†Department of Electrical and Computer Engineering
Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu - www.cs.utexas.edu/users/cart

ABSTRACT

Pipeline flushes are becoming increasingly expensive in modern microprocessors with large instruction windows and deep pipelines. Selective re-execution is a technique that can reduce the penalty of mis-speculations by re-executing only instructions affected by the mis-speculation, instead of all instructions. In this paper we introduce a new selective re-execution mechanism that exploits the properties of a dataflow-like Explicit Data Graph Execution (EDGE) architecture to support efficient mis-speculation recovery, while scaling to window sizes of thousands of instructions with high performance. This distributed selective re-execution (DSRE) protocol permits multiple speculative waves of computation to be traversing a dataflow graph simultaneously, with a commit wave propagating behind them to ensure correct execution. We evaluate one application of this protocol to provide efficient recovery for load-store dependence speculation. Unlike traditional dataflow architectures which resorted to single-assignment memory semantics, the DSRE protocol combines dataflow execution with speculation to enable high performance *and* conventional sequential memory semantics. Our experiments show that the DSRE protocol results in an average 17% speedup over the best dependence predictor proposed to date, and obtains 82% of the performance possible with a perfect oracle directing the issue of loads.

Categories and Subject Descriptors: C.1.3[Processor Architectures]: Other Architecture Styles – Scalable Selective Re-execution

General Terms: Measurement, Performance, Design, Reliability, Experimentation

Keywords: Mis-speculation recovery, selective re-execution, dependence prediction, EDGE architectures, microarchitecture, speculation, speculative dataflow machines

1. INTRODUCTION

Faster clocks, deeper pipelines, wire delays, and larger instruction windows necessitate reducing the cost of mis-speculation recovery. The number of mis-speculations increases with increasing numbers of in-flight instructions as predictors in high-ILP processors become more aggressive. Furthermore, the growing cost of on-chip communication is likely to increase speculation recovery latencies appreciably in the future. Finally, as on-chip communication becomes more expensive, a plethora of microarchitectural structures must make decisions with less information, requiring more and more predictors scattered throughout the microarchitecture. Thus, the number of predictors, the number of mispredictions, and the cost of each misprediction are all likely to increase, forcing future processors to spend larger fractions of execution time recovering from mispredictions.

Selective re-execution (SRE) is an increasingly popular technique, in which only the instructions dependent on a violation must be re-issued on a mis-speculation. For example, both the Alpha 21264 [15] and the Pentium 4 [12] use limited SRE to recover from load scheduling mis-speculations. We believe that scaling SRE in conventional processors will become progressively more difficult, due to the following three challenges:

- Tracking and maintaining dependences between large amounts of in-flight state
- The complexity of having many predictions in flight from multiple, distributed heterogeneous predictors
- The increasing physical distance between the distributed detection of violations and the centralized recovery control

In a recent study of various current and proposed re-execution schemes, Kim et al. [16] conclude that "universal selective replay, where an instruction can cause a recovery event at any point during its lifetime, is barely feasible for current-generation designs, and does not scale to wider machines or additional types of speculation."

In this paper, we describe a set of protocols for mis-speculation recovery, that exploit the unique features of a new class of architectures to provide efficient, distributed selective re-execution (DSRE) of data mis-speculations over the entire instruction window. This new class of ISAs, called Explicit Data Graph Execution (EDGE) instruction set architectures, permit limited dataflow-like execution within defined program regions, and conventional execution across those regions, with sequential memory semantics and a conventional programming model. The explicit representation of the dataflow graph in the EDGE ISA obviates the dynamic reconstruction of data dependences in the processor.

We use the EDGE-based TRIPS architecture as the evaluation platform for DSRE. The distributed instruction window in the TRIPS architecture reduces complexity, by facilitating re-execution without requiring the instructions to be buffered in a centralized location, necessary for conventional microarchitectures. The protocol that we propose provides a simple mechanism that multiple heterogenous predictors can use for mis-speculation recovery, while also being scalable for both increasing instruction window sizes and wire delays.

The DSRE protocol enables multiple "waves" of speculative execution to traverse the dataflow graph simultaneously. To ensure that the right answer is eventually produced and committed, a "commit wave" traverses the DFG behind the waves of speculative execution and ensures that the correct results are eventually saved. In this paper, we propose two techniques to accelerate the commit wave, which can become the bottleneck in this scheme. We use DSRE to increase the performance with one type of speculation, namely load-store dependence speculation. Our results show that letting all loads issue aggressively and using DSRE to recover can outperform the best competing dependence predictor by 17%, and achieve 82% of the upper-bound performance of perfect load/store speculation.

The DSRE protocol, which enables lightweight recovery from load/store order violations, is not limited to dependence prediction recovery. The same protocol can be used to recover from *any* data value mis-speculation–including other types of value predictors, such as last-value predictor or stride-value predictor–as well as recovery from soft errors. Since the DSRE protocols we propose use only point-to-point messages to implement recovery, they are ideal for distributed microarchitectures built in future technologies, and may be an enabling technology that support new types of speculation or execution on highly unreliable computational substrates.

In Section 2, we further discuss how efficient mis-speculation recovery will be a growing challenge for future large-window machines, as the number of speculation mechanisms increases. We also analyze load-store dependence speculation showing that current dependence predictors will be ineffective for future distributed microarchitectures. In Section 3, we provide a brief review of the simulated architecture and provide pointers to other literature on EDGE ISA-based machines. In Section 4, we describe the base DSRE protocol, and quantify the performance of DSRE without commit acceleration. We describe and measure commit acceleration techniques in Section 5, and also study the sensitivity of DSRE to larger instruction window size and higher network delay. Finally, we conclude in Section 6 by discussing the possibilities exposed by merging data speculation with EDGE-based dataflow execution.

2. THE NEED FOR EFFICIENT MIS-SPECULATION RECOVERY

The trends toward higher-ILP processors, coupled with steadily increasing on-chip communication latencies [1, 17] and issue window sizes, threaten to make mis-speculation recovery the dominant component of performance loss. Many researchers are proposing issue windows of hundreds or thousands of instructions. As the number of instructions in flight increases, so does the amount of state that must be cleared on a pipeline flush. As Akkary et al. observe, waiting until a mis-speculating instruction reaches the head of the reorder buffer simplifies recovery, but can increase the latency significantly for larger windows [2]. To reduce flush latency at the expense of complexity, many modern processors perform a rolling flush, in which all instructions younger than the mis-speculating instruction are cleared as soon as the misprediction is detected.

As the number of speculation techniques in modern processors grow, efficient mis-speculation recovery becomes increasingly important. Some candidate speculation mechanisms for future microprocessors include:

- Load-store dependence prediction
- Different types of data value speculation like last-value speculation and stride-value speculation
- Predicate prediction
- Control independence speculation [11]
- Coherence speculation in multiprocessors [13]

In this paper, we evaluate the use of DSRE to reduce the mis-speculation penalty for load-store dependence speculation. Issuing loads out-of-order with respect to stores is necessary for high ILP in current and future machines. Current machines use load/store dependence prediction to facilitate early issue of loads. However, effective load speculation is growing more difficult for several reasons. First, larger instruction windows mean that more conflicting load/store pairs will exist in the window, putting more pressure on the dependence predictors. Second, the cost of flushing the pipeline upon a misprediction is increasing as the in-flight state increases and control becomes more distributed. Third, the performance losses due to dependence mispredictions become more of a bottleneck as ILP elsewhere is increased. Fourth, since wire delays will force partitioning in future architectures, dependence predictors are likely to be distributed along with cache banks, reducing their accuracy.

2.1 Maintaining Sequential Memory Semantics

In out-of-order processors, sequential memory semantics must still be maintained. Program-earlier stores must forward their values to later loads for correct execution. The *conservative* policy for addressing this issue is to prevent a load from issuing until all earlier stores with unresolved addresses have issued. The ideal policy is an *oracle*, in which loads that do not conflict with earlier stores issue to the caches but wait for the conflicting store if a conflict exists.

Microarchitects have tried to approximate oracle performance by providing dependence predictors, which allow some loads to issue in the presence of earlier unresolved stores, speculating that they will not be dependent, and flushing the pipeline if incorrect. Loads incorrectly predicted to be dependent on an earlier in-flight store do not cause a pipeline flush, they merely lose an opportunity for higher performance by issuing late, after previous stores are resolved even though they could have been safely issued earlier.

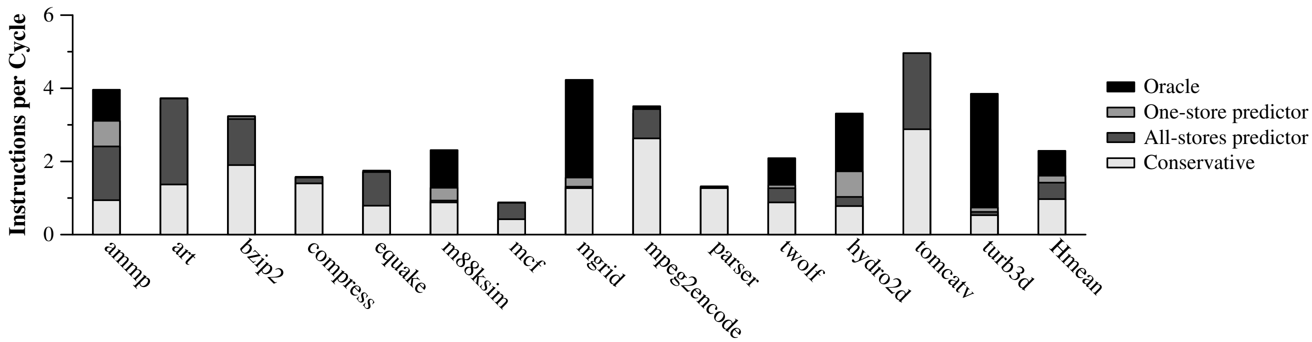

Figure 1: Performance effects of load/store ordering policies with a 2K window

A few examples of simple dependence predictors include those proposed by Moshovos et al. [20], which used simple PC-based indices into tables of saturating counters that specified whether or not a load should issue speculatively, and store-wait tables as implemented in the Alpha 21264 [15], where a load-on-conflict prediction waits for all previous stores to resolve. Store sets [6] are a more complex proposal that attempts to match up loads with specific stores, so that potentially dependent loads do not have to wait for *all* previous stores to resolve, just the ones likely to conflict. Finally, Yoaz et al. [31] propose a predictor that uses distance estimations to approximate store set capabilities with reduced complexity. In this paper, we simulate two types of dependence predictors, *all-stores* and *one-store*.

All-stores: This strategy is similar to the predictor organization of Moshovos et al. [20]. This predictor uses the PC to index tables with 4K 1-bit saturating counters, that are set on a conflict, and only predict no conflict for a load when the counter value is zero. When *all-stores* predicts a conflict, a load waits until all prior stores complete before issuing safely. The table is cleared unconditionally every 10,000 blocks. PC-indexing outperformed the other indexing functions we measured, including address and PC-address hybrid indices, as well as the less aggressive store-wait tables of the Alpha 21264. This predictor matches the predictor being implemented in the TRIPS prototype [3].

One-store: The second type of predictor we simulate is a modified variant of store sets [6]. Modifications were necessary because the distributed architecture that we simulate cannot enforce issue order among stores. Thus, we modified the predictor, which we call *one-store*, to force a load to wait for exactly one store, rather than a set of stores as in the original proposal. The next paragraph details the differences between our *one-store* scheme and store sets.

The *one-store* predictor uses a PC-indexed Store Set Identifier Table (SSIT) to maintain a common tag for each load and store pair. These tags, called Store Set Identifiers (SSID), are stored in the Last Fetched Store Table (LFST), and are described in detail in the original paper [6]. Initially these tables are empty and all loads are predicted as non-conflicting. When a load-store ordering violation is detected, a SSID index is allocated to the violating load-store pair, and an entry is created in the SSIT for the load and the store containing this SSID. During block dispatch, all stores in the dispatched block access the SSIT table to check for a valid LFST entry. If a store finds a valid entry, it inserts its instruction identifier in the corresponding entry in the LFST table. Loads in a block also index the SSIT table during dispatch. When a load finds a valid SSIT entry, it checks the LFST table for a valid store entry. If a load finds a valid store in the SSID table, it marks itself as being dependent on the store. When a load resolves and reaches the memory interface, it checks to see if it has been marked as being dependent on a store. If the load is marked dependent, it sends data back to its consumers only after the pertinent store arrives at the memory interface. In our experiments, we used a 4K entry SSIT table and a 128 entry SSID table. The SSIT table was indexed using the last 12 bits of the PC and was unconditionally cleared every million cycles.

We used a Trimaran/TRIPS-based simulation environment for evaluating DSRE performance. The experimental methodology is explained in greater detail in the next section. Figure 1 shows the performance of the simulated processor using conservative load/store issue, *all-stores*, *one-store*, and oracular prediction. These experiments assumed the TRIPS prototype configuration of 64 frames, which corresponds to a 1K issue window across the 16 ALUs. The graph confirms prior results that the conservative policy performs poorly with respect to the oracle policy, which is 2.37 times faster on average. The *all-store* dependence predictor improves performance significantly over the conservative approach, but only by 46%, which is approximately only one third of the additional performance improvement obtained by the oracle policy. The more aggressive *one-store* predictor performs much better (66 %) but still achieves only a fraction of what is possible with *oracle*, due to the performance lost by flushing the pipeline on a mis-speculation.

2.2 Memory Speculation for Large Windows

As issue windows grow larger, from hundreds to thousands and eventually tens of thousands of instructions, the number of potential conflicts (stores followed by loads to the same address) grows. This growth threatens to limit the parallelism that can be exploited in future high-ILP machines.

Table 1 details the load/store conflict behavior for four different window sizes (1K-8K instructions). These experiments assumed a perfect branch predictor, so the window is always filled with useful work unless the processor pipeline is being flushed from a load/store mis-speculation. We also allow a maximum of 256 hyperblocks to be resident concurrently. The three IPC columns show the average instruction throughput for three of the four ordering schemes from Figure 1. The column labeled *Potential Conflict* shows the fraction of loads in the instruction window that reference

Window Size	IPC			Potential Conflict	Dependence Predictor Performance (% Accesses)			
	Conservative	Oracle	one-store		PD:ED	PD:EI	PI:EI	PI:ED
1K	1.17	3.74	2.15	12.17	17.04	12.17	71.57	1.03
2K	1.19	4.87	2.23	14.58	19.19	13.64	68.26	0.92
4K	1.19	5.39	2.23	17.10	21.04	15.02	65.35	0.76
8K	1.19	5.69	2.32	19.37	21.61	15.85	64.21	0.55

Table 1: Performance characterization with memory dependence prediction for a 4K store sets predictor

Benckmarks	Potential Conflicts (as % of total loads)						
Window Size	128	256	512	1024	2048	4096	8192
mgrid	0.50	1.27	1.85	0.49	0.23	1.25	3.89
applu	0.42	1.33	0.47	0.16	0.22	1.22	13.18
vpr	21.37	24.55	30.17	35.16	39.69	43.30	44.24
art	1.12	1.20	0.99	1.23	1.34	1.73	2.33
mcf	3.61	11.77	19.73	23.14	31.31	30.46	25.42
equake	9.44	25.04	31.61	34.10	32.91	33.03	32.95
ammp	8.38	9.30	8.77	9.59	10.16	12.08	10.77
parser	14.19	18.43	23.10	24.89	25.17	27.46	34.28
eon	11.07	21.03	30.20	38.95	39.59	44.25	44.36
perlbmk	14.94	21.88	26.33	29.44	33.03	39.54	45.13
gap	3.63	8.17	14.72	23.18	25.99	22.68	23.38
gzip	12.98	20.84	25.93	13.72	10.49	15.27	11.08
bzip2	15.61	21.05	38.60	40.35	40.20	40.28	40.40
Average	9.02	14.30	19.42	21.11	22.33	24.04	25.49

Table 2: The number of potential conflicts with increasing window sizes with perfect branch prediction

the same address as a store that is also in the window, when using oracle load/store dependence prediction. Note that a conflict will actually occur only if the load issues out of order from the store. Not surprisingly, the fraction of potentially conflicting loads increases with window size, and combined with a larger number of loads in the window, results in a much larger total possibly conflicting loads. The remaining columns show the behavior of the *one-store* predictor. For large instruction windows (8K), on average 15% of the predicted accesses are predicted as dependent (PD) and actually end up independent at execution time (EI), thus increasing the load latency for these accesses. Fewer than 0.6% of the accesses are predicted independent (PI) and are actually dependent (ED), requiring a rollback recovery. The remaining 85% of the loads have their dependence predicted correctly and incur no penalty. From column 7 in Table 1, we can see that the predictor becomes increasingly conservative as windows size increases, and a greater percentage of the loads are incorrectly predicted to conflict, unnecessarily forcing them to wait. This class of loads stands to benefit greatly from DSRE.

To validate the output of the Trimaran/TRIPS-based simulation, Table 2 shows the number of potential conflicts observed from a sim-alpha based simulation with perfect branch prediction [7]. While the benchmarks from the SPEC suites that we were able to run are somewhat different, the overall trends are the same.

The remainder of this section describes the related work from the scientific and patent literature in the area of selective re-execution implementations for centralized architectures. Section 4 then describes a class of distributed selective re-execution policies that provide large speedups over basic load/store dependence prediction.

2.3 Previous Work on Selective Re-Execution

Many researchers have explored and are exploring selective re-execution to defray growing mis-speculation costs. Some of the earliest work was done by Rotenberg et al. [25], who discussed applying selective re-execution to both control and data mis-speculations recovery for Trace Processors. Selective re-execution for control prediction exploits control independence [26], and can be used for techniques like out-of-order fetch [28].

Calder et al. [4] showed that selective re-execution coupled with dependence prediction can–in a centralized microarchitecture with small issue windows–approach the performance of a perfect dependence predictor. Those techniques are insufficient to provide the same gain on distributed microarchitectures with much bigger (1000+ entry) instruction windows, which is the problem that we address.

Despite its potential benefit, implementation complexities prevent current selective re-execution schemes from being used as a single unified recovery mechanism for multiple types of data value speculation. Recent patents from AMD [14], Sun Microsystems [22], and Intel [18, 19] propose selective re-execution for recovering only from load scheduling speculation, using signals from the lower-level cache [22] or circular queues [18, 19] to facilitate the re-execution schemes. Multiple disparate recovery modes are used due to the design complexity introduced by interaction among distinct types of speculation, complexity which is exacerbated by slowing global wires. Slowing communication is causing multi-cycle delays between misprediction detection and reporting, which will grow progressively worse if speculation resolution remains centralized. Ernst et al. [8] also made this observation in recent work. The selective re-execution schemes we propose in this paper do not suffer from those challenges, since they provide a single, dis-

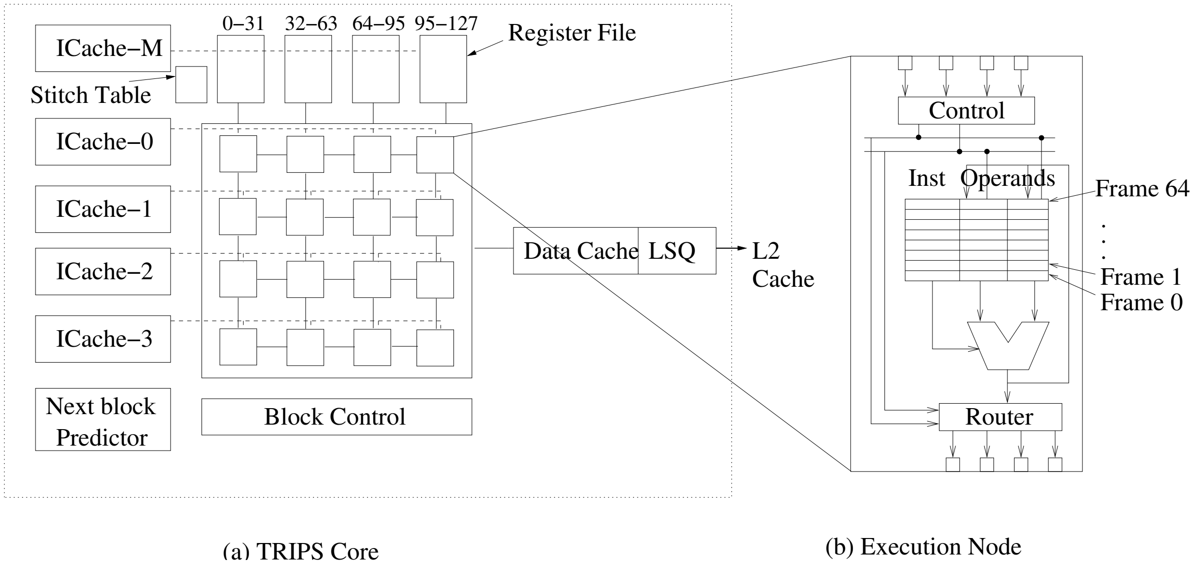

Figure 2: Simulated 4x4 TRIPS-like core

tributed framework for handling potentially many types of speculation simultaneously.

Researchers have also investigated compiler-assisted approaches for efficient memory disambiguation. Gallagher et al. [10] proposed the memory conflict buffer for memory disambiguation. In their approach, the compiler aggressively hoists load instructions above store instructions and inserts correction code to provide recovery, when there is an address conflict. The memory conflict buffer is used for detecting these conflicts. This scheme relies on a centralized issue queue for initiating recovery, and is thus unsuitable for distributed architectures. Early load address computation using compiler support has been investigated by Cheng et al. [5]. However, their approach requires changes to the ISA so that the microarchitectures can differentiate between the various types of loads in the system.

Finally, Zhou et al. [32] identify the challenges associated with implementing aggressive selective re-execution on a conventional superscalar processor, which include retention of issued instructions that may be re-executed, the reissue mechanism itself, and the data-dependence driven identification of the set of instructions to be re-executed. Of the solutions they describe, the ROB augmentation that holds instructions in the window until committed is most similar to the DSRE protocols proposed in this paper. However, their approach is not scalable to larger windows and distributed microarchitectures, nor does it eliminate the performance losses associated with their proposed solutions to the other two challenges.

3. EXPERIMENTAL METHODOLOGY

To measure the performance losses associated with memory ordering violations and the performance of selective re-execution schemes, we simulate a TRIPS-like processor, which is an instance of an EDGE architecture [3, 27]. The selective re-execution schemes described later in this paper rely on the block-atomic execution model and the dataflow-like execution of this architecture.

The specific processor simulated in this paper, shown in Figure 2, is a 16-wide issue machine in a 4x4 configuration, driven by executables generated by the Trimaran [30] compiler, that are re-translated for the simulated processor core. These cores use an EDGE (Explicit Dataflow Graph Execution) instruction set, in which the physical location of the consumers, called *targets*, are encoded directly into the bits of each producer instruction. The compiler predicates basic blocks of instructions and combines them into hyperblocks, each of which have only a single entry point. The blocks execute according to a Static Placement, Dynamic Issue (SPDI) execution model; instructions in each hyperblock are statically assigned to individual ALUs by the compiler, but execute in dataflow order. Thus, the processor places the instructions to minimize wire delays, but the hardware executes the instructions in dynamic order, achieving higher instruction-level parallelism than fully static issue can typically achieve.

The block-atomic execution model fetches one block at a time, maps it onto the execution array, and commits the entire block at once when complete. Thus, instructions do not commit individually, but in blocks when everything in the block has finished. By using a branch predictor to predict the next block, multiple hyperblocks can execute simultaneously on the processor substrate, with all but one executing speculatively. The issue window is distributed and doubles as a "block reorder buffer"; its size is the number of instruction reservation stations at each ALU, times the number of ALUs. We refer the reader to the literature for more detailed descriptions of this execution model [21, 27].

The processor configuration that we simulate in this paper resembles the TRIPS prototype implementation [3]. The processor has 16 fully functional ALUs, and they are connected in a mesh network that consumes 1 cycle per hop. There are 64 reservation stations per ALU and the processor can map 1024 instructions at one time. A maximum of 8 hyperblocks can be resident on the processor concurrently. The branch predictor is a hyperblock exit predictor loosely modeled on the Alpha 21264 tournament predictor [24].

The simulated memory system has 32KB L1 instruction

C–Code

```
for (j=1; j < 10; j++)
   a[0] = a[0]+j;
```

TRIPS Assembly

```
loop_body:
  Read G[0] N[0, 0] ;   Read j
  Read G[1] N[2,0] N[4,0] ;  Read a[0] address
  N[0] addi 1 N[1,0] ;  j = j+1
  N[1] mov N[3,0] N[5,0]
  N[2] lw 0 N[3,1] ; Load a[0]
  N[3] add N[4,1] ; a[0] = a[0] + j
  N[4] sw 0 ; Store a[0]
  N[5] mov N[6,0] W[0] ; write j to register
  N[6] addi 1 N[7] ; Increment loop count
  N[7] teqi 10 N[8] ; Check for loop termination
  N[8] mov N[9,P] N[10,P] ; Move predicte bits
  N[9] bro_f  loop_body ; if false, jump to loop_body
  N[10] bro_t exit ; Exit if true
```

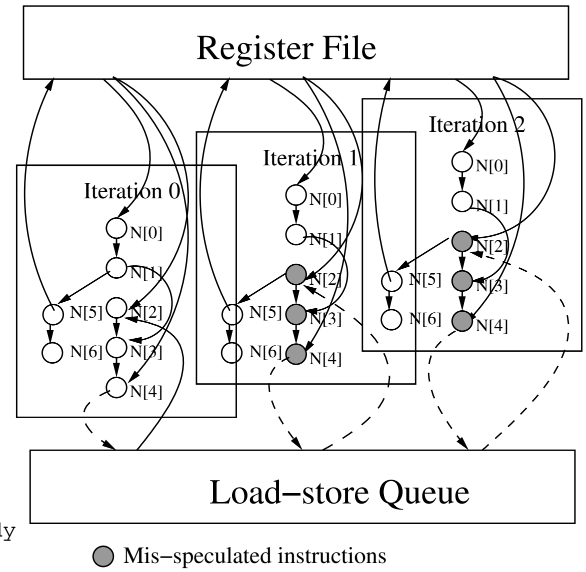

Figure 3: Re-execution on an EDGE architecture

caches partitioned into four banks, and a centralized 32KB data cache and load-store queue. The TRIPS prototype system has distributed data caches and a physically distributed, logically centralized LSQ. We intend to validate the performance of the DSRE protocol using the TRIPS prototype high-level model in the future, as our current Trimaran-based simulator does not support distributed data cache and load-store queue. The 2-way L1 instruction and data caches each have 64-byte blocks and 1 and 2-cycle hit times, respectively. The L2 cache is 2-way set associative, with a 12-cycle hit latency and a 132-cycle miss penalty to main memory. The baseline dependence predictor is a centralized 4K table of 1-bit saturating counters indexed by the PC. We simulate a set of 14 SPEC95 (compress, hydro2d, tomcatv, turb3d, m88ksim), SPEC2000 (ammp, art, equake, bzip2, mcf, mgrid, parser, twolf), and mediabench (mpeg2encode) benchmarks. We used the set of benchmarks that the Trimaran tools were able to compile successfully in our environment. For each benchmark, we fast-forwarded through the initialization phase and simulated 100 million instructions. In the next section, we use this simulated processor model to evaluate the proposed DSRE protocol.

4. DISTRIBUTED SELECTIVE RE-EXECUTION

EDGE architectures lend themselves to efficient, distributed selective re-execution (DSRE). In the TRIPS instantiation of an EDGE ISA, instructions and their operands are buffered as they arrive at the reservation stations. When an operand arrives, its tag indicates the reservation station and instruction operand to which it corresponds. When all necessary valid bits for an instruction's operands have been set, the instruction fires, executes, and sends the result to its consumers, which are specified using one or two target fields in the just-issued instruction.

The multiple hyperblocks in flight effectively form a large dataflow graph (DFG). Within hyperblocks, the DFG is a statically constructed graph, with arcs going from ALU to ALU. Cross-block arcs are instantiated through register names; each block reads from and writes to a subset of the architectural registers. If a hyperblock produces an output allocated to R3, and the subsequent hyperblock requires an input read from R3, the value of R3 will be forwarded from the older to the younger block as soon as it is produced. Thus, the large DFG is a collection of smaller, statically produced DFGs stitched together by dynamically resolved cross-block arcs through the register file and inter- and intra-block arcs through memory.

Figure 3 show the C-code for a simple loop, along with the corresponding TRIPS assembly code. We also show the data flow graph for 3 different iterations of the main loop body in the figure. The code snippet has two loop carried dependences, one through memory and one through the register file (register 0). The loads in successive iterations of the loop depend upon the store in the previous iteration. The loop carried dependence through memory is shown by dashed lines. As shown in the figure, this dependence is enforced by the load-store queue. If a load mis-speculates, the DFG sub-tree of the load gets incorrect values. These nodes are shown shaded in the figure. In a conventional implementation without selective re-execution, a mis-speculation will trigger a flush of all instructions after the violating load.

To initiate DSRE of an instruction that has computed with a wrong value, the correct value is simply sent to the incorrect instruction's node with the same tag as the original incorrect instruction. As shown in Figure 3, the instruction re-fires, sending a new output value to its dependent children. The children subsequently re-fire, and so on, eventually re-executing the entire DFG subtree data dependent on the faulting instruction. A re-fired instruction producing a value that crosses hyperblock boundaries sends a newer version of its result to the target hyperblock, which will cause additional instructions to re-fire. Note that this model permits low-complexity DSRE without having to re-issue any instructions not dependent on the erroneous instruction, nor do any instructions need to be re-fetched, re-dispatched, or moved. Assuming that the ALU and network contention

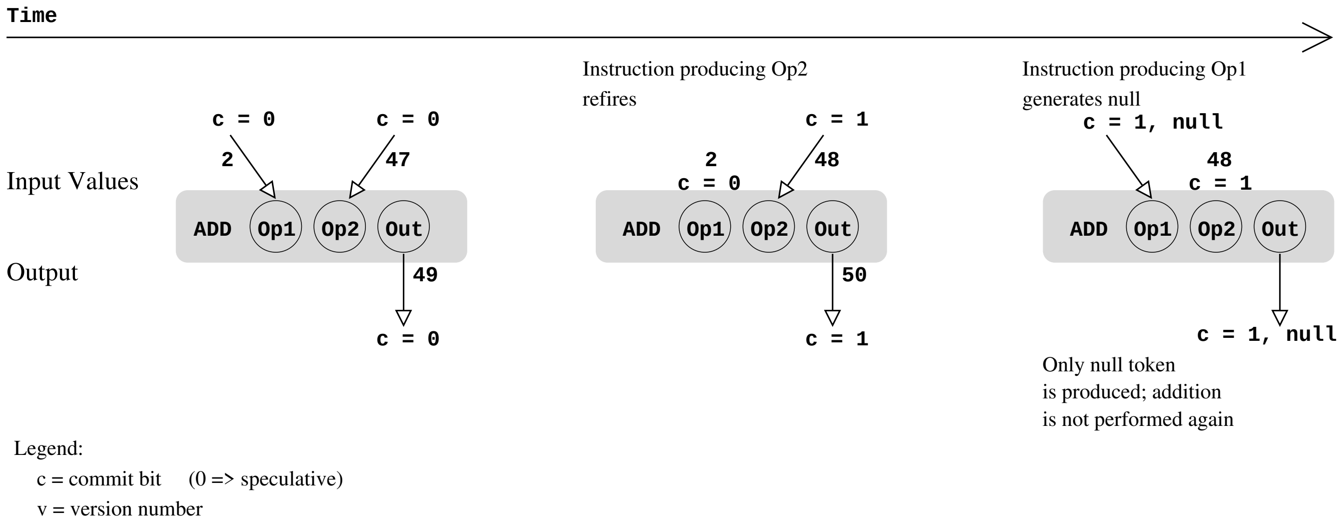

Figure 4: Illustration of commit messages

and extra energy consumed by the re-execution are insignificant, re-execution in an EDGE ISA is always beneficial, since re-executing an operation and everything dependent on it is no worse than having waited for the actual correct value rather than speculating.

4.1 Commit Waves: Detecting Block Completion

In the TRIPS processor, when each hyperblock completes, it is removed from the array and its instructions are all committed at once; hyperblocks logically commit atomically. A hyperblock's stores are written back to the memory system in order, which happens during hyperblock commit. Since the TRIPS processor is a distributed microarchitecture, detecting completion of the oldest hyperblock is accomplished with point-to-point messages. The block header of each hyperblock contains a count of all hyperblock outputs (register writes, stores, and a single branch). When an output instruction in a hyperblock reaches the register banks or caches, the output counter for that hyperblock is decremented. When it reaches zero, all of the outputs of the hyperblock have fired, meaning that the block is safe to commit if it is the oldest hyperblock. If stores or writes are on predicated paths through the hyperblock, the compiler inserts *null store* or *null write* instructions on the other paths, so that the block always produces the same number of outputs.

However, this scheme for detecting block completion cannot work without modification in the presence of re-execution policies. Since multiple waves of execution may be traversing the hyperblock's DFG simultaneously, the output instructions may receive their source operands multiple times, and thus do not know when it is safe to signal the completion logic that they have completed non-speculatively.

The solution we explore in this paper is to add a *commit bit* to each valid bit at the instructions' operand buffers. When a commit bit is set, it signals that its operand is no longer speculative, and none of that operand's parents in the DFG may be speculative. The commit bit is only zero if there are still unresolved data speculations among its parents. Note that the control speculation (branch prediction) mechanisms are separate from these data speculation techniques. In order for a block to commit, it must be the control non-speculative block and all of its register and store outputs must be non-speculative.

When all of an instruction's operands have received their commit bit, then the result computed using those operands is also non-data-speculative. We simulate two types of commit bit messaging, shown in Figure 4. First, if an operand arrives at an ALU with its commit bit set, and the instruction's other operands are also non-speculative, then the instruction fires (or re-fires) and sends its result to its consumers with its commit bit set in the message control header. The second case occurs when an instruction has already fired–and has already sent its result with a zero commit bit–but is later determined to have been correct and becomes non-speculative. In this case, a *null commit message* is sent to the consumers of that instruction, signaling that the operand previously sent is now non-speculative.

This late determination of correctness may happen for several reasons. An arithmetic operation may compare its speculative, buffered operand with the receipt of a committed operand, and if they are the same, the result need not be recomputed, but a null commit message can be sent. More commonly, a load may have issued speculatively, in the presence of earlier unresolved stores (with a zero commit bit). When the load's address has received its commit bit, and *all* earlier stores have also received their commit bits–and if there was no address conflict with a store–then the load becomes non-speculative and a null commit message may be sent.

A block is thus safe to commit when it is the oldest block (guaranteeing that there is no more control speculation) and when all of its outputs have received their commit bits. In terms of the DFG, the process of detecting completion can be thought of as a "commit wave" traversing the DFG behind the dataflow execution, and signaling completion when traversal of a hyperblock's portion of the DFG is complete.

The addition of commit bits requires small extra hardware capability : one bit of state for inter-ALU operand message headers, some additional control to encode and decode null commit messages, three extra bits per instruction reservation station state, and some extra logic in the load/store queues to support committing loads on the commit bits, not the valid bits, of earlier stores.

4.2 Version Numbers: Out-of-Order Messaging

The scheme described thus far allows multiple, partially or fully overlapping waves of speculative execution traversing the DFG, succeeded by a "clean-up" commit wave. This model is simple so long as multiple speculative versions of an operand are always injected into the inter-ALU network in order *and* the network supports in-order delivery of messages. If either of those requirements are not met, then the possibility of overwriting the correct computation with later-arriving mis-speculative data arises. For example, assume that an instruction fires twice and produces versions A and B, where B is later determined to be correct, so is followed by a null commit message. If A and B are re-ordered (either by injection into the network or by the network itself), then the instruction's consumers will receive B, fire correctly, then receive A, fire incorrectly, and then receive the null commit message saving the incorrect result computed with A.

This case would occur if the network re-ordered messages, although the network we simulate does not exhibit this problem, because routing is deterministic and messages are never dropped. However, another window of vulnerability is opened by the possibility of injecting speculations in the wrong order. For example, assume a load with earlier unresolved stores accessed the cache but missed. Subsequently, a program-earlier store to the same address issued, so the store value is forwarded to the load and sent to the load's consumers (version B). The mis-speculated cache access eventually returns and could be forwarded to the load's consumers (version A)– overwriting the correct computation triggered by version B. This case could be avoidable by adding extra support to the memory system, but serves as an illustrative example.

We handle out-of-order messaging by augmenting transmitted operands with version numbers as well as commit bits. Each arc of the DFG can be traversed multiple times, with its version number increasing with each of its source operands. For example, if an ADD instruction fired three times, the version numbers of the operands sent to its consumers would be 0, 1, and 2, in order, regardless of what the version numbers of the ADD's source operands were. The highest version number is always the correct operand, and null commit messages are tagged with the version number that they are committing.

This scheme permits operands to be re-ordered and still function correctly, since version numbers are buffered with the operands and commit bits at the consuming instruction's reservation station. If an ADD instruction has fired twice because it received two values of its left operand, which arrived with version numbers 0 and 2, and then later a version of the left operand arrives with version number 1, that message is discarded because that operand has already received a higher version number, guaranteeing that the lower one is incorrect. If a null commit message arrived with a version number 3 for the left operand, the instruction would wait to receive the actual operand tagged with version number 3 before re-firing and propagating the result to the consumers with the commit bit set. This policy permits operands and commit bits (whether arriving with an operand or as null commit messages) to arrive in any order but still produce correct execution and guaranteed completion. We show an example in Figure 5.

INCOMING MESSAGE	STATE O1/V/C	O2/V/C	ACTION
01=2, v =0, c=0	2/0/0	–	No Action – only one operand has arrived.
02=42,v=0, c=0	2/0/0	42/0/0	Add inputs and send output message (out=44; v = 0, c= 0). The output is speculative because inputs are speculative.
02=49, v=3, c=0	2/0/0	49/3/0	Add instruction is re−executed and a new result is sent outwith a new version number. The results are still speculative.(out=51, v =1, c=0)
01=null, v=0, c=1	2/0/1	49/3/0	operand 1 is marked as non−speculative. New output is not generated b/c the operand values have not changed.
02=null, v=4, c=1	2/0/1	49/3/1	Generate null token; both inputs are non−speculative.
02=45, v=2, c=0	–	–	Message is dropped because its version number is lower than the last versionnumber received for this operand.

Table Legend:
STATE: Output value/Version number (V)/Commit bit (C)

Figure 5: Version number example

Since the counters that track version numbers will be finite, the system must ensure that it does not overflow them. For an N-bit version counter, $N-1$ speculative versions may be generated, but the Nth must be guaranteed to be correct. Therefore, after generating $N-1$ speculative versions, an instruction waits until it has all of its sources' commit bits before producing the Nth version. It may, of course, discover that version $N-1$ was correct and just send a null commit message instead.

With commit bits, completion of distributed selective re-execution can be detected, and with version numbers, the computation will still be correct in the presence of reordered messages. While we have used load/store dependence prediction as the driving example for this protocol, *any* data speculation scheme may use this underlying framework for low-overhead recovery, so long as it obeys the rules of the protocol: The last version sent is always the correct one (with no versioning support), or the highest version number is always the correct one (with versioning support). Thus, many types of data value speculation may make use of this common framework for low-overhead recovery.

Version numbers also provide a convenient mechanism to throttle speculation. ALUs can be prevented from firing speculatively when the version of their result reaches a certain maximum value. To find this optimal value, we performed experiments varying the maximum version number allowed. Table 3 shows the mean performance across our benchmarks for the best load-store policy, when we vary the maximum allowed version number from one to six. We can see from Table 3 that performance decreases progressively as we increase the maximum version number allowed. This indicates that the best performance is obtained when nodes are allowed to fire only once or twice speculatively. We restricted the maximum version number to one in the rest of our experiments.

Version number max value	1	2	3	4	5	6
Mean IPC	1.884	1.876	1.873	1.875	1.861	1.849

Table 3: Mean performance for various maximum version numbers

	No flush		Flush on load mis-spec		Flush on commit mis-spec		Ideal		Bypass
Benchmark	**cons**	**DSRE**	**all-stores**	**one-store**	**all-stores**	**one-store**	**p-com**	**oracle**	
ammp	0.94	1.52	2.41	3.11	3.27	3.84	3.96	3.96	3.86
art	1.37	1.89	3.72	3.50	3.72	3.72	3.73	3.73	3.42
bzip2	1.90	2.14	3.16	3.23	3.19	3.19	3.24	3.24	3.19
compress	1.40	1.55	1.56	1.56	1.65	1.64	1.66	1.66	1.64
equake	0.79	1.20	1.71	1.71	1.74	1.74	1.75	1.75	1.73
m88ksim	0.88	1.10	0.93	1.28	1.15	1.40	2.26	2.31	1.45
mcf	0.42	0.79	0.87	0.83	0.88	0.85	0.88	0.88	0.85
mgrid	1.27	1.68	1.31	1.56	3.52	3.36	4.15	4.23	3.40
mpeg2encode	2.63	3.12	3.43	3.32	3.49	3.46	3.51	3.51	3.47
parser	1.27	1.30	1.31	1.31	1.31	1.31	1.32	1.32	1.31
twolf	0.88	1.11	1.27	1.36	1.72	1.63	2.04	2.09	1.62
hydro2d	0.78	1.34	1.03	1.73	2.87	2.94	3.35	3.35	2.95
tomcatv	2.88	3.82	4.96	4.95	4.96	4.95	4.96	4.96	4.94
turb3d	0.53	0.72	0.62	0.74	0.91	1.00	3.28	3.85	1.01
Mean	0.97	1.36	1.42	1.61	1.84	1.88	2.27	2.30	1.89

Table 4: Performance of Load/Store Recovery Schemes

4.3 DSRE Performance

Table 4 shows the performance of the simulated machine with all of the load/store speculation policies we evaluate in this paper. The target machine simulated is described in Section 3. Performance is displayed in instructions per cycle (counting useful, non-overhead, committed instructions only). We assumed that flushes are rolling, initiated when a misprediction is first detected, which is a higher-performance assumption than initiating flushes when the block containing the faulting instruction is ready to commit.

Column two (the leftmost data column) shows performance using conservative ordering (*cons*), in which every load waits for all prior stores to complete. As we showed in Section 2, this conservative model is by far the worst-performing model, and greatly inhibits ILP. The third column shows performance with a pure re-execution protocol (*DSRE*), in which all loads issue as soon as they are ready, and re-execute if an earlier store resolves to the same address. Pure DSRE provides a 40% performance boost over conservative load-store ordering, making it a potential alternative to dependence prediction. The difference in performance between the DSRE and the oracle policy is primarily due to the commit wave falling behind the execution wave.

Columns 4 and 5 show the performance of traditional dependence prediction, using *all-stores* and *one-store* to selectively stall loads that are predicted to be dependent, and flush the pipeline if a load is speculatively issued before a conflicting store. *all-stores* shows almost exactly the same average performance as DSRE. The more complex, but more aggressive, *one-store* policy improves performance over the base case by an additional 13%, since some stalled loads can proceed earlier when their conflicting store arrives, instead of waiting for all stores. Despite these relatively large performance gains, a large gap still exists with the upper-bound performance of an oracle, which shows a mean IPC of 2.30, 43% faster than the *one-store* policy.

In the next section, we describe new policies that attempt to close this gap, the results of which are shown in columns 6 though 8 and explained after the discussion of those policies. We also examine the scalability of the DSRE protocol for larger window size and higher network latency.

5. ACCELERATING AND SCALING DSRE

This section examines two policies to accelerate propagation of commit bits in order to improve performance with DSRE. We also study the scalability of DSRE, by looking at the performance for larger instruction window and higher network latency.

5.1 Accelerating Commit of Re-executed Blocks

Our results have shown that the commit traversal of the DFG is the single largest remaining impediment to achieving performance close to that of an ideal oracle. Column 8 of Table 4, shows the performance of DSRE with ideal commit performance (*p-com*). In the *p-com* policy, every load issues as soon as it reaches the memory interface, resulting in multiple speculative waves when a store arrives. However, the commit bits in the policy are infinitely fast, so that the commit traversal never inhibits performance. The mean IPC for *p-com* is 2.27, which is within 4% of the upper bound, demonstrating that the commit traversal is the remaining bottleneck. If the commit traversal can be made sufficiently fast, the performance losses due to load/store conflicts will be negligible. In the rest of this section we describe two techniques for accelerating the commit traversal: *speculative commit slicing* and *bottom-up commit traversal*.

5.1.1 Speculative Commit Slicing

Our analyses have shown that a significant portion of the commit traversal's lag behind the execution traversal of the DFG is attributable to late-committing stores. For example, if a store address depends on a load that incurs a L2 cache

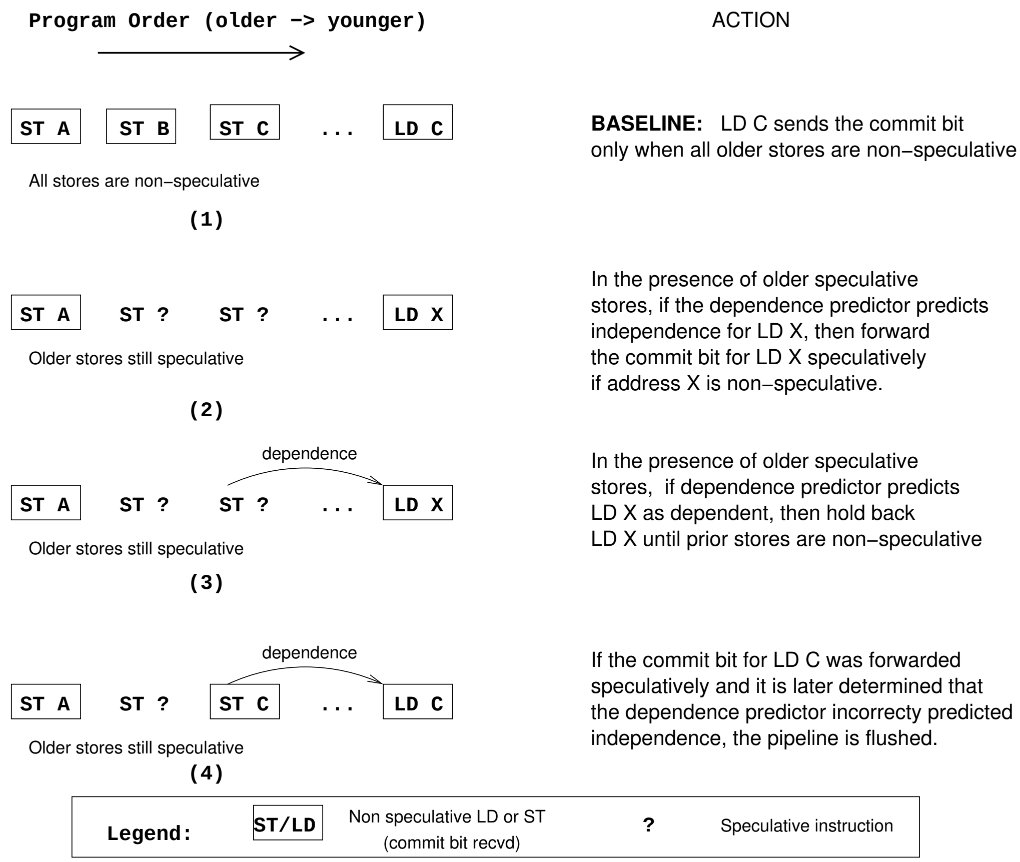

Figure 6: Commit Slicing

miss, and is followed by four loads in program order, the store does not get its commit bit until after the load miss returns. Only then can those loads forward their commit bits to their consumers (provided, of course, that the commit bits for the loads' addresses have also been received). A single slow store can thus block all subsequent loads from forwarding any commit bits until quite late. Since loads typically reside at the head of dependence chains, a single slow store may thus block any significant advance execution of the commit wave.

To accelerate the commit traversal, we allow some loads to forward their commit bits speculatively–although no modifications are made to architectural state until safe commit is guaranteed. A load that is unlikely to conflict can forward its commit bit, and if no violation eventually occurs, the commit bit speculation improves performance. If a conflict does occur, the pipeline needs to be flushed, since there is no way to recall the commit bit. This strategy is safe because no architectural state is written until all commit bits are received, at which point any violations will already be known.

This policy thus uses a hybrid of *selective re-execution* for the aggressive execution of loads and speculation with *flushing* for acceleration of commit bits. To issue the speculative commit bits accurately, we re-employed the dependence predictors evaluated earlier (*all-stores* and *one-store*).

We show an example in Figure 6. If the load is predicted independent, the load sends its commit bit as soon as it receives a commit bit from its address, despite the presence of earlier unresolved or uncommitted stores. If a conflict is later detected, the pipeline must be flushed to guarantee correct execution. If the load is predicted to be dependent, then the load waits until all previous stores have received their commit bit (and its address has sent its bit) before forwarding its commit bit to its successors.

We measured the performance of speculative commit slicing using both dependence prediction strategies, shown in Columns 6 and 7 of Table 4. Using the simpler *all-stores* predictor to perform commit slicing provides a 30% speedup over using it to perform speculative load issue. It also provides a 14% speedup over pure dependence prediction using the more complex *one-store* predictor. Using the *one-store* predictor to do commit slicing, however, provides a smaller 17% speedup over using it for load speculation. Commit slicing with DSRE is faster than using dependence prediction for loads on every benchmark we measured. Commit slicing provides a larger speedup for the *all-stores* predictor, achieving close to the performance of the more complex *one-store* predictor with commit slicing. This is because the *all-store* predictor is more conservative, and hence predicts a larger fraction of the loads as conflicting. This class of loads benefit greatly with DSRE, because only commit bits need to be sent for these loads when they resolve. Thus, DSRE coupled with a simple predictor can be used to achieve performance comparable to that with a more complex predictor.

5.1.2 Bottom-up Commit Traversal

If all operations–including loads–could execute in a single cycle, selective re-execution would provide no benefit over conservative load/store ordered execution, because the commit DFG traversal would take the same time as the execution traversal. DSRE improves performance because not all operations require a single cycle, especially cache misses, so the commit traversal can catch up to the execution traversal while long-latency operations on the critical path execute. However, since no execution actually occurs on the commit wave, it may be possible for the commit wave to skip nodes in the graph, thus completing more quickly.

Speculative Commit Slicing essentially removes some arcs from the commit traversal graph speculatively, allowing more of the graph to be traversed in parallel and speeding up the traversal. An alternate approach is to allow commit bits to skip over nodes, going directly from the input to the output of a multi-instruction dependence chain without traversing the intermediate nodes. If the root of a dependence tree has only one speculative input, then the intermediate nodes in the tree can be bypassed when the last committed operand arrives, by sending the commit bit directly to the leaves, provided no execution is still in flight.

Bottom-Up Commit Traversal selectively allows a partial bottom-up traversal to support forwarding of commit bits over multi-hop chains. If a leaf node–in this case an output-producing instruction–of the DFG has only one speculative parent (all other parents, if any, have sent their commit bits), then the output node forwards its target(s) to the one speculative parent. The output node knows the parent's reservation station address since it has already received an operand from that parent, assuming the address was buffered. When the parent generates a commit bit, it bypasses the intermediate node and sends the commit bit directly to the output, as shown in Figure 7. Prior to committing, however, if the parent has only one speculative parent, it too can forward the output address to its parent (the grandparent), which can then either do the same thing (forward up the chain if it has one speculative parent) or send the commit bit to the output, bypassing two nodes. If an instruction holding a bypass target re-fires instead of generating a commit instruction, then the bypass chain is discarded and the new operand is forwarded to the children as in the base architecture. When the execution reaches the outputs, they can begin the process of rebuilding the bypassing links anew.

The last column in Table 4 shows the performance of this bottom-up traversal scheme when combined with speculative slicing. The bottom-up traversal scheme performs well for some benchmarks due to commit acceleration, while it performs worse on others due to the extra network traffic. The mean performance of this scheme is marginally better than speculative slicing with the *one-store* policy. However, the bottom-up traversal scheme incurs significant hardware complexity over the base DSRE scheme, and is not worth the marginal performance improvement. We are exploring additional hardware support, such as a separate commit network, as a possibility for further acceleration of commit bits.

5.2 DSRE Scalability

To study the scalability of the DSRE protocol, we increased the instruction window size to 2K instructions and, in a separate experiment increased the network delay to two cycles. We show the results of these experiments in Table 5. We show the performance of the base configuration–a 1K window and one cycle inter-ALU delay–in the third row.

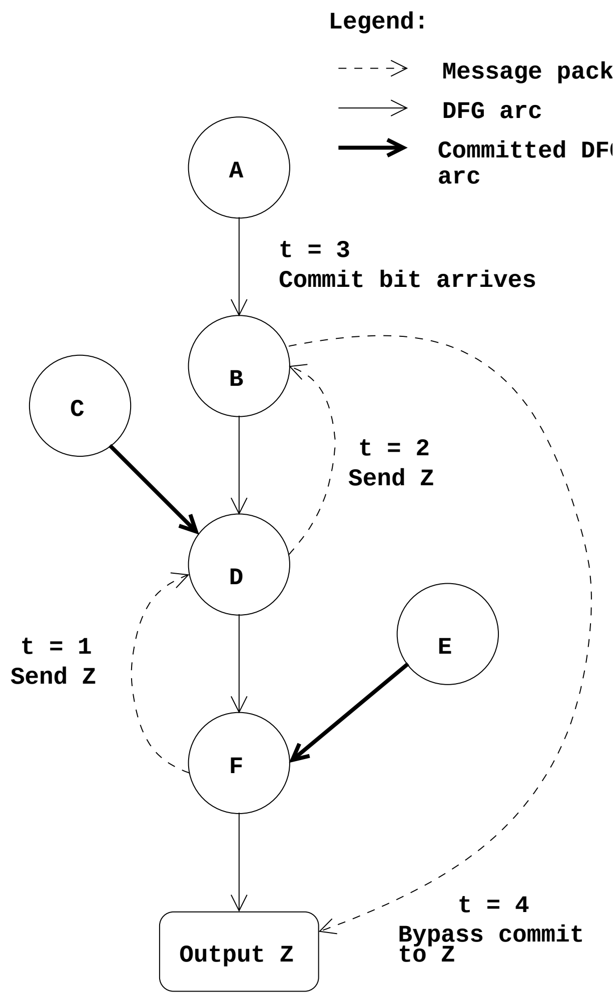

Figure 7: Bottom-up Traversal

The fourth row in Table 5 shows the mean IPC with various load issue policies with a 2K window. When the window size is doubled, the performance improves by a mere 2% with a conservative load-issue policy, demonstrating the well-known result that load speculation is necessary to exploit large-window ILP. The oracle policy improves by 26%, showing the potential performance advantages of scaling the window size. Conventional dependence prediction (with flushing) improves by just over half of the ideal, increasing by 14% using the *one-store* policy. The SRE implementations scale much better with increasing window size; *one-store* policy with DSRE results in a 25.5% improvement in performance, and *all-stores* improves by 27%. DSRE with commit slicing thus scales similarly in performance to the oracle as the window size grows, even as conventional dependence prediction (with flushing) tails off.

The last row in Table 5 shows the mean performance for the various load issue policies with a two-cycle network latency. In this configuration, DSRE with speculative commit slicing achieves 78% of the performance of a perfect oracle, which is less than the performance improvement obtained

	No flush		Flush on load mis-spec		Flush on commit mis-spec		
Configuration	**cons**	**DSRE**	**all-stores**	**one-store**	**all-stores**	**one-store**	**oracle**
Base	0.97	1.36	1.42	1.61	1.84	1.88	2.30
2K window	0.99	1.86	1.63	1.84	2.35	2.36	2.90
2-cycle network	0.78	0.96	1.14	1.28	1.35	1.42	1.82

Table 5: Mean Performance of Load/Store Recovery Schemes

in the base 1-cycle network latency case (82% of oracle). However, the degradation of performance is fairly consistent (20-24%) across all policies. DSRE is thus ideally scalable with increasing window size, and moderately scalable with increased network delay.

6. FUTURE IMPLICATIONS OF DSRE

For machines that implement EDGE instruction sets, of which the TRIPS architecture is one example, we have shown how to design a fully distributed selective re-execution protocol that requires only simple, local state machines.

DSRE protocols such as these will provide future distributed microarchitectures with low-overhead recovery from value mispredictions. In this paper, we focused on load/store ordering speculation, using a DSRE protocol to show a 17% performance improvement over conventional dependence predictors.

The processing of the commit tokens, not ALU or network contention, caused the most performance losses in the DSRE protocol. We evaluated one technique (speculative commit slicing) that achieved 82% of the performance of an oracle predictor, and proposed and evaluated a bottom-up commit graph pre-traversal for hiding parts of the commit graph traversal. This technique, however, did not result in performance improvements large enough to justify the additional hardware complexity.

Although we focused mainly on load/store dependence speculation, DSRE protocols can easily handle other types of value speculation, including value prediction, predicate prediction, and even "physical speculation," executing instructions on ultra-fast or ultra-low-energy ALU that may occasionally produce a wrong answer but has physical benefits in the common case [9]. Looking further ahead, DSRE protocols can support low-overhead recovery for both misspeculations and certain types of soft errors. After all, an operation in a fault-susceptible system is just a correctness speculation, which cannot be committed until re-executed and checked.

Speculative Dataflow Machines? Branch mispredictions still cause enormous performance losses in high-end processors, and branch predictors are improving with only diminishing returns. While some other architectural proposals advocate moving to a "more pure" dataflow model [29] that has little control, they merely shift the control dependences to data dependences that must be executed conservatively, both in registers and memory, placing a tight asymptote on achievable parallelism.

DSRE protocols can enable a different solution in emerging EDGE architectures–the compiler grows enormous hyperblocks to control-flow graph merge points, which encompass any control flow splits and merges. Within these large, predicated blocks, a predicated producer of a value may choose to fire speculatively and inject its operands to the rest of the graph. If the actual needed operand should have been generated on a different path, the correct operand can simply be re-injected and handled gracefully by the DSRE protocol. The execution of predicates can thus be removed from the critical path by speculating the values of certain predicates, with a low-overhead, DSRE-supported recovery guaranteed if the predicate was mispredicted.

The EDGE architecture model with huge hyperblocks, little explicit control flow, and a fine-grained dataflow ISA, starts to resemble in many aspects past dataflow machines like Monsoon [23], but with one important distinction: the dynamic changing of dataflow arcs can be supported by forwarding values into the DFG speculatively and aggressively, with the DSRE protocol providing a clean recovery if wrong. Monsoon also had multiple functional units connected by a dynamic network, with each functional unit having a token-store for receiving tokens. This token-store was made explicit in the data flow model to simplify resource management. However, dataflow arcs in Monsoon were fixed, as it did not have support for speculation. Data speculation, coupled with distributed selective re-execution, may eventually make dataflow architectures truly competitive by also allowing them to achieve high performance on conventional, imperative languages.

7. ACKNOWLEDGMENTS

We thank Professor Anant Agarwal for his comments on an initial draft of the paper. This research is supported by the Defense Advanced Research Projects Agency under contract F33615-03-C-4106, NSF instrumentation grant EIA-9985991, NSF CAREER grants CCR-9985109 and CCR-9984336, two IBM University Partnership awards, grants from the Alfred P. Sloan Foundation and the Intel Research Council.

8. REFERENCES

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate vs. ipc : The end of the road for conventional microprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 423–434, December 2003.

[3] D. Burger et al. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

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

[5] B.-C. Cheng, D. A. Connors, and W. mei W. Hwu. Compiler-directed early load-address generation. In *Proceedings of the 31st Annual ACM/IEEE*

International Symposium on Microarchitecture, pages 138–147, December 1998.

[6] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proc. of the 25th Annual Int'l Symp. on Computer Architecture (ISCA'98)*, pages 142–153, June 1998.

[7] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proc. of the 28th Int'l Symp. on Computer Architecture*, pages 266–277, June 2001.

[8] D. Ernst and T. Austin. Practical Selective Replay for Reduced-Tag Schedulers. In *Proceedings of the 2nd Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD-2)*, pages 58–63, June 2003.

[9] D. Ernst, N. S. Kim, S. Pant, S. Das, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 7–18, December 2003.

[10] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages (ASPLOS-VI)*, pages 183–193, October 1994.

[11] A. Gandhi, H. Akkary, and S. T. Srinivasan. Reducing branch misprediction penalty via selective branch recovery. In *Proceedings of The Tenth International Symposium on High-Performance Computer Architecture*, pages 254–264, December 2004.

[12] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[13] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: Making use of incoherence. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages (ASPLOS-XI)*, October 2004.

[14] J. B. Keller, R. W. Haddad, and S. G. Meier. Scheduler which discovers non-speculative nature of an instruction after issuing and reissues the instruction. United States Patent 6,564,315, May 2003.

[15] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.

[16] I. Kim and M. Lipasti. Understanding scheduling replay schemes. In *Proceedings of The Tenth International Symposium on High-Performance Computer Architecture (HPCA'04)*, pages 138–147, December 2004.

[17] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, September 1997.

[18] A. A. Merchant, D. J. Sager, and D. D. Boggs. Computer processor with a replay system. United States Patent 6,163,838, December 2000.

[19] A. A. Merchant, D. J. Sager, D. D. Boggs, and M. D. Upton. Computer processor with a replay system having a plurality of checkers. United States Patent 6,094,717, July 2000.

[20] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.

[21] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 40–51, December 2001.

[22] R. Panwar and R. C. Hetherington. Appartus for executing coded dependent instructions having variable latencies. United States Patent 5,987,594, November 1999.

[23] G. Papadopoulos and D. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 28–31, May 1990.

[24] N. Ranganathan, R. Nagarajan, D. Burger, and S. W. Keckler. Combining hyperblocks and exit prediction to increase front-end bandwidth and performance. Technical Report TR-02-41, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, September 2002.

[25] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors . In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 138–148, December 1997.

[26] E. Rotenberg, Q. Jacobson, and J. E. Smith. A study of control independence in superscalar processors. In *Proceedings of The Fifth International Symposium on High-Performance Computer Architecture (HPCA'99)*, pages 115–124, January 1999.

[27] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, S. W. Keckler, D. Burger, and C. R. Moore. Exploiting ilp, tlp and dlp with the polymorphous trips architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[28] J. Stark, P. Racunas, and Y. N. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *International Symposium on Microarchitecture*, pages 34–43, December 1997.

[29] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 291–302, December 2003.

[30] Trimaran : An infrastructure for research in instruction-level parallelism. http://www.trimaran.org.

[31] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proc. of the 26th Annual Int'l Symp. on Computer Architecture (ISCA'99)*, pages 42–53, May 1999.

[32] H. Zhou, C. ying Fu, E. Rotenberg, and T. Conte. A study of value speculative execution and misspeculation recovery in superscalar microprocessors. Technical report, ECE Department, N. C. State University, January 2000.