*Appears in the 1st Value-Prediction Workshop*

# Lightweight Distributed Selective Re-Execution and its Implications for Value Speculation

Rajagopalan Desikan* Simha Sethumadhavan Ramadass Nagarajan 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

*In this paper, we describe a lightweight protocol to support selective re-execution on the TRIPS processor. The protocol permits multiple waves of speculation to be traversing a dataflow graph simultaneously and in any order, with a cleanup "commit" wave propagating as well to determine completion of a group of instructions. The protocol is completely distributed, consisting of point-to-point messages and requiring no centralized control. Thus, recovery from value mis-speculations requires no additional fetching or decoding of instructions, and no issue of instructions that were independent of the faulting instruction. We describe briefly one way in which this protocol can be exploited: by allowing* every *instruction to use a decentralized last value predictor. Our results show that in this scheme up to 26% of all instructions can fire as soon as they are fetched, with 0.001% of instructions firing incorrectly.*

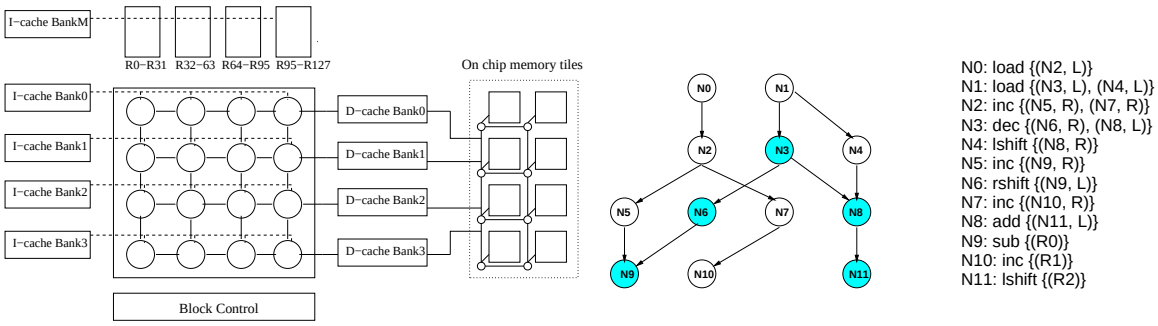
## 1 Introduction

Speculation has become significantly more widespread and aggressive as processors have gone to deeper pipelines, wider issue, and higher frequencies. While control speculation has historically been the main focus of speculation research, data speculation has grown in research popularity over the past 5 years. However, there are three major problems with current implementations of data speculation. First, in some processors, rollback is quite expensive, triggering a pipeline flush. This will only be exacerbated in future deeply pipelined, wide issue machines with a large number of instructions in the pipeline. Second, more recent processors, such as the Alpha 21264 [11] in a limited fashion, and the Pentium IV [8] in an aggressive fashion, implement *selective re-execution* in which only instructions dependent upon the faulting instruction are re-executed. In centralized superscalar processors with deep pipelines, these selective re-execution policies cause an explosion in control and datapath complexity, and are unlikely to scale to higher ILP machines. Third, the implementation of proposed data predictors has problems; implementing multiple predictors is complex, as they may compete for speculations and have unintended interactions. Furthermore, centralized predictors make it difficult to get a high bandwidth of predictions out of the predictor, especially if the rate of instructions accessing the predictor is high.

In this work, we describe the selective re-execution protocol that we are currently implementing, that is intended to solve the problems enumerated above. This selective re-execution protocol is a completely distributed re-execution protocol, based on the Grid Processor Architecture [19], which permits ultra-low-overhead rollback from data mis-speculations by incorporating efficient selective re-execution. This protocol permits multiple speculations to be in flight simultaneously, permitting more aggressive speculation than conventional re-execution protocols. The protocol has a simple design, while permitting arbitrary number of speculation engines to inject speculations, and can even support multiple speculations of the same operand in flight at once. Finally, the implementation of selective re-execution is simple and distributed, requiring no centralized control or state.

To make the complexity feasible, we envision a single speculation engine that is run by multiple speculation state machines. The challenge in that implementation is in discovering the right way to have different instructions select the state machines that are best suited for the type of access they need to do.

We illustrate the potential of this single speculation engine by employing a data predictor scheme in which every instruction, arithmetic or memory, that produces




a) GPA b) Selective Re−execution

**Figure 1. 4x4 Grid Processor Architecture.**

register value is permitted to speculate in a distributed fashion. We show that with simple control, 26% of all instructions that are fetched can produce their result correctly at decode, with a negligible number of misspeculations, from which the selective re-execution protocol can recover efficiently.

In the rest of this paper, we first provide a review of the block-atomic Grid Processor Architecture, which is the hardware substrate on which we are implementing these value predictors. In the following section, we describe the distributed selective re-execution (DSRE) policy. We then describe a fine-grained last-value predictor implementation, in which every instruction is fetched along with its last value and highly confident instructions fire immediately upon fetch.

## 2 Block-Atomic Execution on Grid Processor Architectures

The Grid Processor Architecture (GPA) [19] is a family of architectures that are designed to scale to high performance at future wire-dominated technologies. It is a "*static-placement, dynamic-issue*" (SPDI) processor that follows a block-atomic execution model. In this model, programs are compiled into large blocks of instructions with a single entry point, no internal transfers of control, and possibly multiple exit points, as found in hyperblocks. Each block has a static set of state inputs, and potentially a variable set of state outputs that depend on the exit point from the block. The compiler statically assigns each instruction to an execution resource and explicitly encodes inter-instruction dependences within a block. At run-time, the execution proceeds by fetching a block of instructions from memory, mapping them onto the execution resources, loading the state inputs, and executing the instructions in dynamic dataflow order. Intermediate values that are produced and consumed within the block are routed directly from producers to consumers without being committed to persistent store. After the block is executed to completion, architecture state is committed. Exceptions, if any, cause rollback to the last committed block boundary.

The hardware is composed of a two-dimensional array of homogeneous execution nodes (ALUs) connected by a routed operand network as shown in Figure 1a. An execution node consists of an integer unit, floating point unit, router ports at the input and output, and a set of reservation stations, each of which contains storage for an instruction and its two source operands. The distributed L-1 instruction cache, data cache, and a register file are placed around the periphery of the execution array. When a reservation station has its instruction and two operands ready, the node can select the instruction for execution. After execution, the node forwards the result along the operand network to the consuming reservation station. When a block has all of its instructions executed, its outputs are committed and its execution resources freed for use by subsequent blocks.

To achieve high performance, the GPA supports aggressive control speculation. The next block predictor speculatively selects the next block that should be fetched and executed, while the current block is still executing. Instructions in speculative blocks can execute as soon as they receive their input operands. To enable this, the GPA also supports aggressive forwarding of outputs from a block to consumers in subsequent blocks. In the current implementation, the GPA achieves 91% hit rates in the next block predictor and can sustain execution of upto 32 hyperblocks concurrently.

Data value speculation is a technique to overcome the limits on parallelism imposed by data dependences in the program. Using data value speculation, an instruction's result is predicted before it is calculated and forwarded to the instruction's consumers. The speculation is later validated when the instruction is actually executed and on a mis-speculation, rollback is initiated. In the next section, we describe a low cost mechanism for

implementing data value speculation on the GPA.

# 3 A Distributed Selective Re-execution Protocol

To support efficient data value speculation, we are implementing a protocol that enables simple, distributed selective re-execution, which is a light-weight mechanism for recovering from data mis-speculation. In the DSRE protocol, the data-flow graph (DFG) is mapped as a block onto the GPA and is traversed by speculative waves of computation. The ALUs receive speculative operands in this scheme, fire, and forward their speculative results to subsequent instructions mapped at other ALUs. On a mis-speculation, only the instructions in the mis-speculated path need to re-fire and propagate the correct data values. By selectively re-executing instructions that received the wrong data values, we can have a low cost data speculation recovery mechanism, thus enabling highly aggressive data value speculation. We show an example of selective re-execution of a dependent sub-tree of a DFG in Figure 1b. The colored nodes are the nodes which need to be re-executed, to correct a data mis-speculation. In this example, instructions scheduled on nodes N6, N8, N9, and N11 are dependent on the instruction scheduled on node N3. If node N3 speculates incorrectly on the value of its result, and then later corrects the speculation, only the instructions scheduled on the these dependent nodes need to be re-executed.

## 3.1 Challenges for a DSRE Protocol

There are two main challenges to the implementation of the DSRE protocol. First, we need to ensure that the correct value of a speculated operand is not overwritten by an incorrect value. Second, every node needs to know when it has received the correct value for its operands. The selective re-execution protocol in the GPA is intended as a general mechanism for supporting different types of data value speculation. This might include aggressive forwarding of load values to consumers, using data value prediction tables at an ALU node to inject speculative values, and injecting speculative values for invalid lines in the cache in a shared memory multiprocessor configuration.

To support selective re-execution, each instruction buffer at an ALU node on the GPA includes a commit bit, a valid bit, and a version number for each input operand. An operand is assumed to be speculative if its commit bit is not set. Results generated using speculative operands are themselves speculative, and are forwarded as such to their consumers.

In this protocol, instructions can fire as soon they receive all their input operands, irrespective of whether the operands are speculative or non-speculative. When a data value speculation mechanism injects speculative values into the grid, it injects them without setting the commit bit. When the speculation is resolved, the operand is re-injected with the commit bit set. A block is deemed safe to commit when all the outputs of the block have been received with their commit bit set. If it can be determined that the speculative execution was correct, then the nodes do not need to re-fire, and only commit bits need to be propagated in the grid.

## 3.2 Version Numbers

There are two factors which contribute to improved performance with selective re-execution. First, we need a mechanism to identify correctly speculated data values to avoid re-computation. Second, the cost to propagate commit bits in the grid should be lower than the cost to propagate operand data values.

To identify correctly speculated operand values, we use a versioning system in the GPA. Version numbers are required in the protocol as messages can be received by a node out-of-order. Every operand has a version number associated with it. This is set to 1 when an operand is generated for the first time in the GPA. When a new value for an operand is generated, it is re-injected with a higher version number. Every time a node receives an operand with a higher version number, it re-fires and generates a higher version number for its output, which is then propagated to the instruction's consumers. For correctness, the protocol guarantees that the non-speculative value for an operand will have the highest version number. When the data speculation is resolved and is determined to be correct, only the commit bit for the result is injected. The version number for the result is not incremented, to communicate that the speculation was correct, thus avoiding re-computation.

When a node receives the commit bit for one of its operands, it checks the incoming version number of this operand against the last received version number for that operand. If the instruction requires two input operands, and one of the operands is still speculative, no output is generated. If the incoming operand is the last non-speculative operand, then the instruction fires and sends a commit bit for its result to all of the instruction's consumers, without incrementing the version number of the result. This can result in a commit wave with the instruction's consumers in turn firing and generating commit bits, once all their operands become non-speculative. The following list enumerates the various cases that can be encountered in this protocol :

- Only one input operand for a two input operand

instruction arrives at a node. No output is generated.

- The second operand arrives at this node. The commit bit for at least one of the operands is not set. The node fires speculatively and generates a speculative output with the commit bit not set.
- One of the operand arrives again at the node with a higher version number. The node fires again and generates an output. If the commit bits for both operands are set, the generated output is non-speculative and has its commit bit set. Otherwise, the generated output is speculative and its commit bit is not set.
- An operand arrives at the node with a lower version number. The operand is ignored and no action is taken.
- Only a commit bit arrives for one of the operands. If the commit bits for both operands are now set, a commit bit is generated for the output and set to the instruction's consumers. If commit bits for both operands are not set, no output is generated.

In Figure 2, we show some examples to illustrate the use of version numbers. The node in the figure represents an instruction with two inputs and one output, mapped onto an ALU. The version number and commit bit status of the input operands and output operand generated by the instruction are also shown in the figure.

In Figure 2a, initially the node receives speculative input operands with version number 1. The node fires and generates a speculative output operand with version number 1. Subsequently, the node receives a higher non-speculative version for one of its operands and fires again generating a speculative newer version of its output. Finally, the node receives the commit bit for its second operand and generates the commit bit for its output. The version number for the output is not incremented to indicate to the consumers of the instruction that the last speculative output generated was correct.

In Figure 2b, we show the case where the node gets incorrect speculative values for both its input operands. In this case when the speculation is finally resolved, the node generates a new version number for the non-speculative output with the correct output value. Figure 2c shows the case where different versions of an operand arrives at a node out-of-order. Version 3 of the first operand arrives before version 2. The node produces a speculative output when version 3 arrives at the node. No output is generated by the node on the arrival of version 2 of the first operand. When the commit bit for the first operand arrives at the node, the node generates only the commit bit for the result and forwards it to the instruction's consumers. Thus, using the versioning system ensures that only the correct value is forwarded by this node to its consumers.

Finally in Figure 2d, we illustrate the case when multiple speculative versions of the input operands arrive at the node followed by the commit bits for these operands. In this case the node does not fire if it receives the commit bits for one of its operands while the other operand is still speculative. The node also does not fire if the the version number of its output exceeds a threshold. Thus, version numbers can be used for throttling speculation. The node finally fires and generates a non-speculative output when all its input operands become non-speculative.

### 3.3 Load-Store Dependence Speculation

One important application for selective re-execution is load-store memory disambiguation. Aggressive forwarding of load values to consumers is necessary for high performance. By using the selective re-execution mechanism, loads can forward their values to consumers while there are earlier unresolved stores. When stores resolve and determine that a load has fired incorrectly, the load can be fired again with the right value to correct the mis-speculation. Thus, unlike most conventional processors that flush the pipeline on a data value mis-speculation, selective re-execution on the GPA ensures that only the instructions in the mis-speculated path are re-executed to correct the mis-speculation.

Researchers have proposed using load-store dependence predictors to aggressively issue loads in the presence of unresolved stores [17, 5, 25]. These predictors use confidence estimators to predict when a load conflicts with a previous store. When a load is predicted non-conflicting with high confidence, it is allowed to access data from the cache and send it to the load's consumers. The speculation is later validated, and if found to be incorrect, mis-speculation recovery is initiated, either by flushing the pipeline or re-executing the instructions in the mis-speculated load's datapath.

We propose using a mixed approach, that combines a load-store dependence predictor and selective re-execution for load-store disambiguation. In this approach, a distributed load-store predictor with confidence estimator is used to predict conflicts between loads and stores in the instruction window of the GPA. Loads predicted to be no conflicting with a high confidence are allowed to send data to their consumers with the commit bit set. If a mis-speculation is detected later, the pipeline is flushed and the load is re-issued. Loads predicted to be conflicting with a high confidence are throttled and not issued till the conflicting store or all prior stores have been resolved. For loads that cannot be

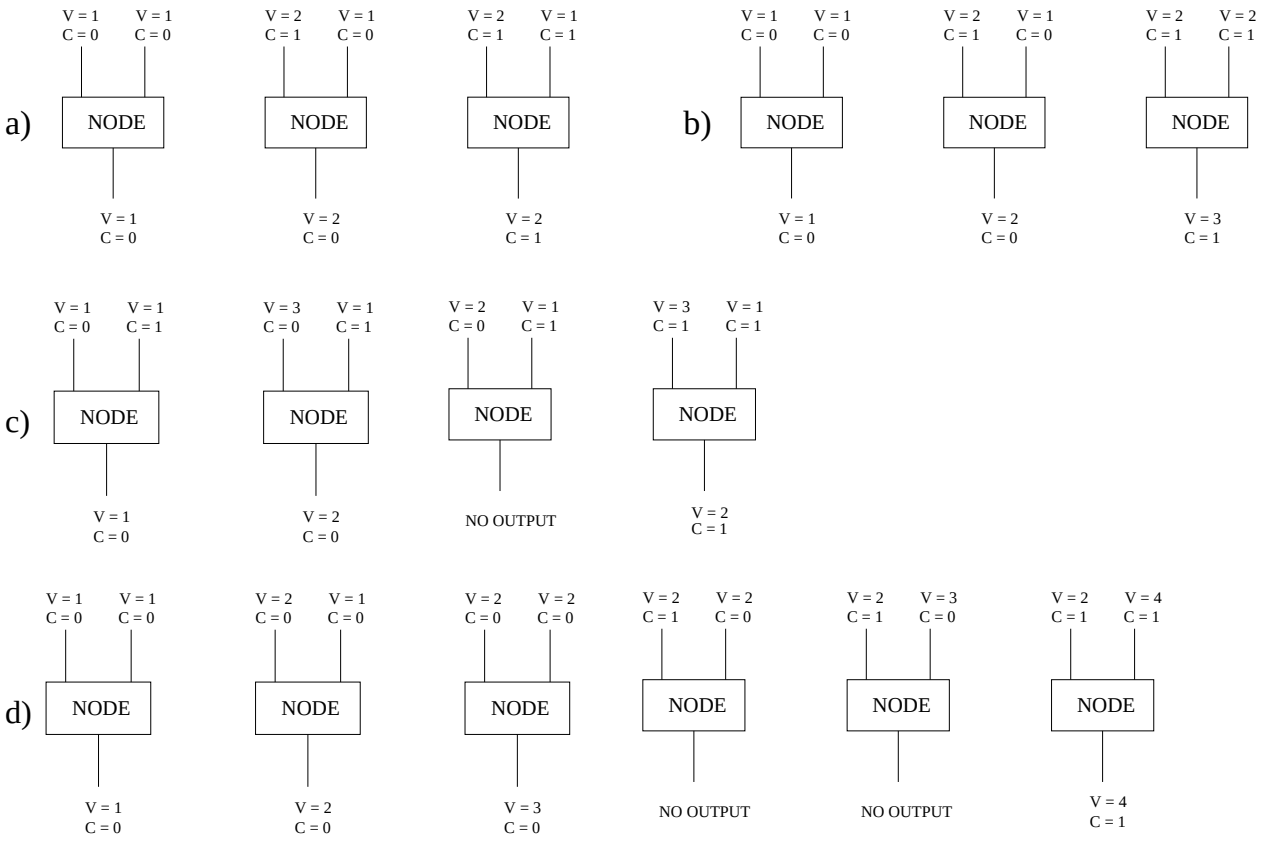



**Figure 2. Version Number Examples**

predicted with a high confidence, we use the selective re-execution mechanism to inject data speculatively and later validate the speculation. Using different load-store predictors, we can tune the correct level of interplay between these mechanisms to obtain optimal performance.

## 4 Decentralized Last Value Speculation

Data value locality and reuse is a phenomena which has recently generated a lot of interest in the computer architecture community. Data value reuse results when an instruction produces the same result during different dynamic invocations. A high data value reuse will result in greater performance improvement with data value speculation.

Value locality was first defined by Lipasti et al. and exploited to perform load value prediction [16]. Using simple predictors, the authors were able to achieve 3% and 6% average improvement in performance on processors modeling the PowerPC 620 and Alpha 21164. Value locality and reuse has been extended in a number of directions since then. Value locality of load instructions has been investigated to eliminate redundancy [24]. The value locality of store instructions has been studied in an effort to reduce multiprocessor data and address bus traffic [14]. Also, a number of predictors have been proposed in literature for predicting values of instructions [21, 23]. Researchers have also examined compiler optimizations for increasing value reuse [2]. Other work in value prediction has shown that considerable instruction fetch bandwidth is needed to speculate on values effectively [7], which is not an issue in our context because of the ultra-high instruction fetch bandwidth provided by GPAs.

To investigate the potential for data value reuse in SPEC CPU2000 programs, we modified the sim-alpha simulator to count reuse for each dynamic instruction executed [6]. We used simpoint simulations and simulated 100 million instructions for each benchmark [22]. We associated 1, 2, and 3-bit saturating counters with each static instruction. Similar confidence estimators have been used by other researchers in earlier work to increase the accuracy of their predictors [16, 20, 23, 1]. We incremented the counter when an instruction's result matched its previous result and decremented the counter

| Confidence Estimator | Throttling Counter | Counter with Poison Bit |
|---|---|---|
| Average | 1.39 | 0.001 |

**Table 1. Percentage of mis-speculated instructions as a fraction of retired instructions**

when it did not. Figure 3 shows the percentage of retired instructions that produced the same result during successive dynamic invocations, for the highest value of the counter associated with the instruction. For brevity, we show the results for only the 2-bit counter in this section.

From Figure 3 we can see that on an average more than 36% of the instructions committed produced the same result in at least four successive dynamic invocations. Thus, these is tremendous reuse in the SPEC CPU2000 suite, which suggests that aggressive data value speculation techniques, along with low cost recovery, have a great potential for performance improvement.

To reduce data value mis-speculation, we associated a poison bit with each static instruction, that is set for an instruction if we mis-speculate. Other related work in value speculation has examined throttling value speculation of instructions that have low confidence, which has a goal similar to the saturating counter and poison bits that we employ [4]. We throttle data value speculation for instructions whose poison bit is set. Figure 3 shows the percentage of retired instructions which showed reuse and which didn't have their poison bits set. We see that even with the poison bit, 26% of the instructions on an average reuse their results.

The benefit of using poison bits is actually reflected in the number of mis-speculations. Table 1 shows the percentage of retired instructions that mis-speculated for the 2-bit counter scheme, with and without the poison bit, averaged across the set of benchmarks. We can see with the poison bit scheme, on an average, around 0.001% of retired instructions mis-speculate. Even without the poison bit, the percentage of retired instructions that mis-speculated is small, at 1.4%. This shows that simple techniques such as using poison bits can be highly effective in reducing mis-speculations, and hence greatly improve the accuracy of data value predictors. By using a low-cost mis-speculation recovery mechanism like selective re-execution, we can further reduce the effect of these mis-speculations on performance.

## 5 Preliminary Results

To investigate the effectiveness of the decentralized last value speculation, we implemented a simple last value predictor in the GPA simulator. The predictor is indexed using the instruction address and stores the last value produced by the instruction. We associated a 2-bit counter with each entry. We incremented the counter every time an instruction produced the same result and decremented it otherwise. The value associated with an instruction is replaced when the high bit of the counter is zero and the counter is reset on a replacement. An ALU speculates on an instruction's result when the high bit of the counter associated with the instruction is 1. We also associated a poison bit with each instruction that is set whenever an instruction mis-speculates. Speculation is throttled for instructions whose poison bit is set. We simulated a set of benchmarks from the SPEC CPU95 suite, the SPEC CPU2000 suite, and the mediabench suite. Decentralized last value speculation was applied to only integer instructions in the benchmarks.

Table 2 lists the performance of the last value predictor across the set of benchmarks. The first column shows the IPC of the benchmark on the base case without value prediction. The second column shows the speedup obtained when using only the 2-bit counter. We see from Table 2 that using the only the 2-bit counter actually hurts the performance of some benchmarks. However, some benchmarks like *adpcm* and *mcf*, show appreciable speedup with the last value predictor. We found that the low accuracy of the 2-bit counter generates a large number of mis-speculated values in the GPA resulting in ALUs firing multiple times to generate the right value.

The third column in Table 2 lists the speedup obtained with the 2-bit counter and poison bit. We see from the table that using a poison bit never hurts performance. Also, some benchmarks like *adpcm* and *twolf* show significant improvement in performance. We found that using the poison bit reduces both the correct and the incorrect value predictions. However, the reduction in the number of mispredictions is far greater than the reduction in the number of correct predictions, thus resulting in either increased performance or no change in performance.

To get higher performance with the decentralized last value predictor, we plan to evaluate a better confidence estimator than the simple poison bit scheme we have used in this paper. We also plan to extend the last value prediction to floating point values to improve its performance.

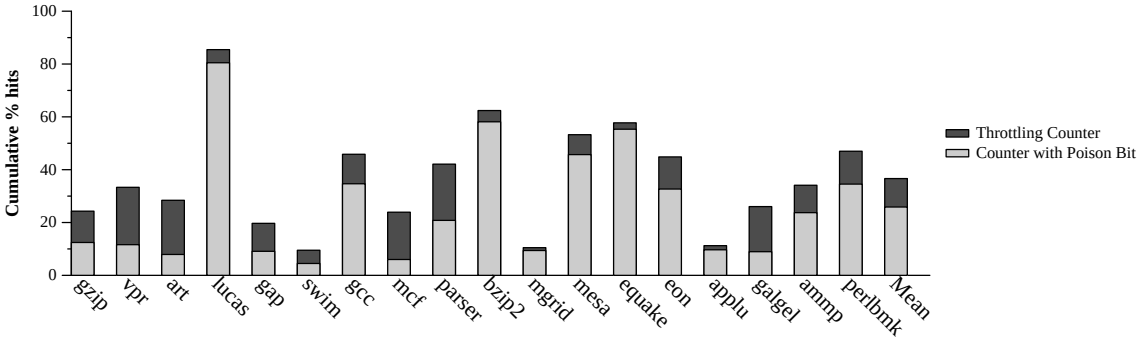



**Figure 3. Correct Value Speculations with Throttling Counter and Poison Bit**

| Benchmark | Base IPC | Speedup - 2-bit Counter | Speedup - 2-bit Counter and Poison Bit |
|---|---|---|---|
| adpcm | 1.3 | 7.7% | 7.7% |
| art | 4.3 | -7.0% | 4.6% |
| bzip2 | 3.6 | 5.6% | 2.8% |
| dct | 7.6 | -1.3% | 0.0% |
| m88ksim | 1.7 | -11.8% | 0.0% |
| mcf | 0.9 | 25.0% | 0.0% |
| mgrid | 2.2 | NA% | 0.0% |
| mpeg2encode | 3.9 | -10.3% | 0.0% |
| parser | 1.7 | -6.2% | 0.0% |
| twolf | 1.7 | 5.9% | 5.9% |

**Table 2. Last Value Speculation Performance on the GPA**

## 6 Related Work

The work most relevant to this proposed work is a paper by Calder et al., in which they compare a number of load-prediction strategies, including dependence prediction, address prediction, value prediction, and memory renaming [3]. This work differs from ours in that the authors did not consider all of the speculation together, determining which policy should be used for distinct loads, nor did they propose a hardware substrate to implement those multiple speculation types effectively. The DSRE protocol enables us to use these prediction techniques at the same time. Finally, that analysis was performed in the context of a rollback scheme that is different from the scalable, low-overhead one that we propose through DSRE.

We do not consider certain other sorts of data speculation in our speculation engine. For example, Lepak and Lipsati [14] analyzed *silent stores*, which do not change the value currently in memory, exploiting them to reduce issued instructions in a uniprocessor, and coherence traffic in a multiprocessor. That work was extended to exploit *temporally silent stores* in multiprocessors [15], in which a value is changed and then changed back, to reduce coherence traffic further.

We also do not consider much of the speculation work that has been done to accelerate coherence protocols without speculating on values. In these protocols, a speculative engine may self-invalidate a block [13], writing it back early so a different node may more quickly gain exclusive access, or protocols that predict coherence traffic patterns to forward shared or exclusive copies of data around the system. Those protocols can be address-based [18, 12] or instruction-based [9, 10]. While these speculation techniques have been shown to have great potential to accelerate programs running on shared-memory machines, they typically are implemented at lower-level caches or memory controllers, and do not interact with rollback mechanisms upon misprediction. A separate engine to unify these techniques where appropriate may be beneficial.

## 7 Conclusions

Many forms of data speculation are now being proposed as key technologies for improving system performance. While prior research on data speculation

have primarily focused on improving prediction accuracies, there has been less work on reducing the penalties associated with mispredictions. In this paper, we describe a new protocol called the *Distributed Selective Re-execution Protocol*, that permits ultra-low-overhead rollback from data mis-speculations by incorporating efficient selective re-execution.

The DSRE protocol we describe uses distributed, simple local state machines, and supports multiple speculation engines, so long as the various engines injecting speculative values obey the protocol. We have shown a speculation engine where up to 26% of fetched instructions can execute immediately with predicted values, with only 0.001% of instructions executing incorrectly. An extremely simple version of this speculation engine was implemented on the GPA and shown to provide appreciable speedup on some benchmarks, with no loss in performance on others. We plan to improve the performance of this engine in future work.

The distributed selective re-execution protocol we describe in this paper permits multiple speculations to be in flight simultaneously, allowing more aggressive speculation than conventional processors. Future work will explore designs that merge different forms of speculation — load value, memory dependence, store forwarding, synchronization (lock elision), and coherence state — into an unified speculation engine, implementing multiple speculative state machines.

## References


[1] M. Burtscher and B. G. Zorn. Exploring last n value prediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 66–76, Oct 1999.

[2] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1:1–6, 1999.

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

[4] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *Proceedings of the 26th International Symposium on Computer Architecture, ISCA-99*, pages 64–74, 1999.

[5] 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, 1998.

[6] R. Desikan, D. Burger, S. W. Keckler, and T. M. Austin. Sim-alpha: a validated execution driven alpha 21264 simulator. Technical Report TR-01-23, Department of Computer Sciences, University of Texas at Austin, 2001.

[7] F. Gabbay and A. Mendelson. The effect of instruction fetch bandwidth on value prediction. In *Proceedings of the 25th International Symposium on Computer Architecture, ISCA-98*, pages 272–281, 1998.

[8] 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.

[9] S. Kaxiras and J. R. Goodman. Improving CC-NUMA performance using instruction-based prediction. In *Proc. of the 5th International Symposium on High Performance Computer Architecture*, pages 161 – 170, Jan 1999.

[10] S. Kaxiras and C. Young. Coherence communication prediction in shared-memory multiprocessors. In *Proc. of the 6th International Symposium High Performance Computer Architecture*, pages 156–167, 2000.

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

[12] A.-C. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Proceedings of the 26th Annual Int'l Symp. on Computer Architecture (ISCA'99)*, pages 172 – 183, May 1999.

[13] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, June 1995.

[14] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 182–191, 2000.

[15] K. M. Lepak and M. H. Lipasti. Temporally silent stores. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 2002.

[16] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[17] 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.

[18] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *Proc. of the 25th Annual Int'l Symp. on Computer Architecture (ISCA'98)*, pages 179 – 190, June 1998.

[19] 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*, December 2001.

[20] G. Reinman and B. Calder. Predictive techniques for agggressive load speculation. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, MICRO-98*, pages 127–137, Dec 1998.

[21] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th International Symposium on Microarchitecture, MICRO-30*, pages 248–258, Dec 1997.

[22] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Archit ectures and Compilation Technique*, Sept. 2001.

[23] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *International Symposium on Microarchitecture*, pages 281–290, Dec 1997.

[24] J. Yang and R. Gupta. Load redundancy removal through instruction reuse. In *International Conference on Parallel Processing*, pages 61–68, 2000.

[25] 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, 1999.