Value Predictors for Reuse through Speculation on Traces

Maurício L. Pilla* and Philippe O. A. Navaux
Computer Science Inst., UFRGS – Brazil
{*pilla,navaux*}*@inf.ufrgs.br*

Bruce R. Childers
Computer Science Dept., Univ. of Pittsburgh – USA
childers@cs.pitt.edu

Amarildo T. da Costa
IME – Brazil
amarildo@cos.ufrj.br

Felipe M. G. França
COPPE/UFRJ – Brazil
felipe@cos.ufrj.br

Abstract

Reusing dynamic sequences of instructions–i.e., traces–improves performance for many benchmarks. However, many traces are not reused because of unavailable inputs in the reuse test. **Reuse through Speculation on Traces** *(RST) aims to increase the number of reused traces by predicting those inputs when necessary, with minimal additional hardware when compared to non-speculative trace reuse.*

In this paper, we compare last n-value and stride-aware prediction for trace inputs. Last n-value prediction uses the last recorded values as predictions, while stride-aware prediction identifies and uses strides to compute new predictions. Stride-aware RST has a higher hardware cost than last n-value RST and has also the shortcoming of not allowing branches inside predicted traces. This paper aims to determine which scheme is the most beneficial for RST. We show that stride values are important for reuse in RST and that last n-value prediction works as well as the more sophisticated stride-aware approach with simpler hardware.

1. Introduction

Control and data dependencies are the main issues that prevent current processors from exploiting more instruction-level parallelism and achieving better performance. It is known that programs execute a large amount of redundant or predictable computations [4, 11, 20, 22]. Value reuse and prediction techniques have been developed to mitigate the effects of dependencies by exploiting redundancy and predictability in programs.

Value reuse is non-speculative. After a block of computations is reused, their outputs can be written without executing the computations again. However, reuse requires that inputs are available (i.e., ready) when execution reaches a reusable computation. The unavailability of inputs can be a severe restriction that limits the performance gain from value reuse, especially for trace reuse.

On the other hand, value prediction is speculative and can provide inputs earlier than they are calculated, which allows instructions with true data dependencies to execute in parallel. However, predictions may be wrong and instructions with mispredicted inputs must be re-executed.

We have previously described a novel technique, called *Reuse through Speculation on Traces* (RST), that combines reuse and prediction in an integrated mechanism [15, 16, 17]. In this scheme, traces that could not be reused by regular trace reuse due to the unavailability of inputs can be reused by predicting values for the missing inputs. RST has the potential for increasing performance by a harmonic mean speedup of 1.43 over regular trace reuse (RST with perfect confidence) [16].

In this paper, we present a comparison of two different value prediction techniques for RST: *last n-value prediction* and *hybrid last n-value and stride prediction*. When using the last n-value mechanism, RST makes predictions based only on previously seen values, which are already stored in memoization tables. Including stride prediction in this mechanism increases the potential for better prediction and more reuse opportunities. Here, new traces can be specu-

* Work partially done with grants from CNPq, CAPES and HP projects

latively created on-the-fly based on the difference between previous consecutive traces (i.e., the input and output values have strides).

This paper is divided as follows. Section 2 introduces the basic operation of Reuse through Speculation on Traces. Section 3 presents different prediction strategies for RST. The experimental environment and results are discussed in Section 4. Section 5 describes previous work and Section 6 concludes the paper.

2. Reuse through Speculation on Traces

Reuse through Speculation on Traces (RST) [15, 16, 17] improves trace reuse and hides true data dependencies by combining value prediction and value reuse in an integrated approach. RST allows traces to be *(i)* **regularly reused**, when all inputs are ready and match the values stored in the input context of a trace; or *(ii)* **speculatively reused**, when there are unknown values in the input context of a trace. Therefore, traces that could not be reused in previous approaches may be reused to exploit their predictability and not only their redundancy.

Although only a small percentage of traces are reused in non-speculative trace reuse, a speedup of 9.5% can still be obtained [15]. Based on this potential, RST was developed to overcome the limitations of regular trace reuse by providing missing inputs. By providing such missing inputs, a larger percentage of traces can likely be reused, which leads to better performance. In [16], we present a limit study for RST with oracle confidence that shows RST has a potential speedup of 1.43 over an architecture with non-speculative trace reuse.

Figure 1 shows the pipeline for a superscalar architecture with RST. Stages for the RST mechanism execute their tasks in parallel to the instruction pipeline, avoiding an increase in hardware complexity along the critical path of instructions. The first stage, **RS1**, fetches reuse candidates from trace memoization tables. The next stage, **RS2**, compares the inputs of a reuse candidate (instruction or trace) to the current values stored in the register file (reuse test). If there are trace candidates whose input values depend on values that have not been calculated yet, RST may predict these values and speculatively reuse traces. The third stage, **RS3**, tracks and verifies predictions. The last stage, **RS4**, identifies reusable instructions and forms new traces.

2.1. Trace creation and reuse

Traces of dynamic instructions are created during runtime based on sequences of instructions in the reuse domain (integer, branches, and the address calculation for loads and stores). Figure 2 shows a static assembly code and how traces are dynamically built from it.

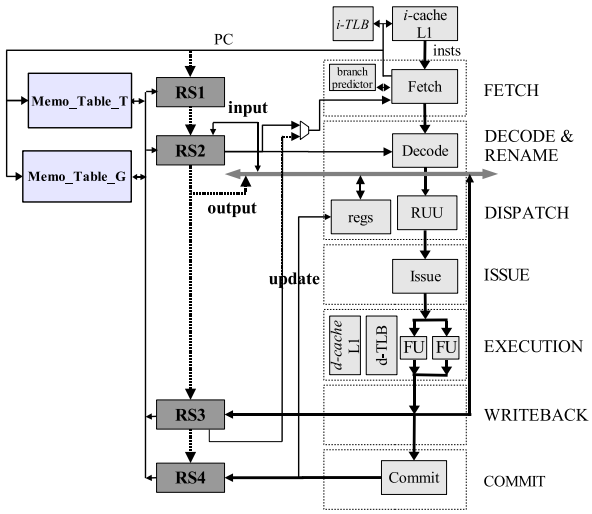

Figure 1. Superscalar pipeline with RST

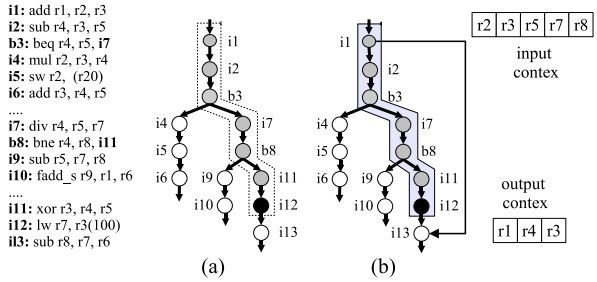

Figure 2. Trace construction in RST

Traces are constructed from sequences of redundant instructions, stored in hardware tables called *Memo_Table_G* (which holds isolated instructions) and *Memo_Table_T* (which holds traces).

As a trace is constructed, its input and output contexts (i.e., the trace live-ins and live-outs) are determined on-the-fly as instructions are added to the trace. The input context is built by tracking whether instruction source register operands are defined inside or outside of the trace. Whenever a source operand is encountered that has not been previously used or defined in the trace, the register operand and its current value are added to the input context. Similarly, destination registers are tracked to identify live-outs. The last value written to a destination register serves as the output value for that register.

Traces are finished by *(i)* instructions outside the reuse domain; *(ii)* lack of resources (size of input/output contexts, number of branches). In addition, if memory access reuse is not allowed, then *(iii)* by loads or stores. In the latter case, only the address calculation is reused. Depending

on the trace creation policy, only reused instructions may be allowed in the trace or instructions not reused may be allowed. The later policy is the default in RST.

After a trace is built and stored, it may be reused the next time that the execution flow reaches its first instruction as in Figure 2 (b). The reuse test verifies if the registers in the input context match the current architecture state. If the test holds, then the output context is written to the registers, the execution flow is redirected to the next instruction after the trace (in this case, instruction *i13*), and branch prediction is updated. Thus, the instructions and data dependencies inside the trace are collapsed.

3. Value Prediction Strategies for RST

In this section, we describe the prediction strategies that we implemented for RST: *Last n-Value Prediction*, where prediction is based only on previously seen values, and *Stride Prediction*, where the predictor can recognize the difference between consecutive traces and predict new values based on this difference. We also discuss the implementation issues related to the use of stride recognition and prediction.

3.1. Last value predictor

One of the simplest implementations of value prediction is the **Last Value Predictor** [4] (Figure 3), which predicts a value based on the last value seen to exploit constant value sequences. The instruction address is used to access the prediction table, where the prediction for the value is stored. The pitfall of this kind of predictor is that it can only correctly predict values when there is one possible and recent value for the input being predicted.

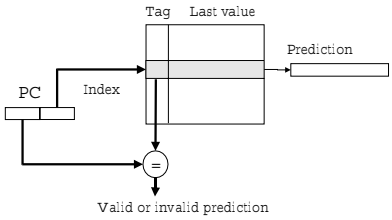

Figure 3. Last value predictor

RST uses a variation of last value prediction where the last n values for an entry may be stored and predicted. If a trace candidate for reuse is found and some inputs are not ready for the reuse test (i.e., the comparison between values stored in *Memo_Table_T* and current values), then the unavailable inputs may be predicted with the stored values. Because it is possible for n trace candidates to exist for a given PC address (where n is *Memo_Table_T* associativity), one trace has to be chosen for speculation. RST selects this trace based on least-recently used information kept with the trace (but other heuristics could be used to select traces).

3.2. Stride predictor

In the last value predictor, only values already seen can be used as a prediction for the next value. However, this mechanism may be extended to allow the dynamic creation of traces based on both the initial values seen and the subsequent differences–called "strides"–between consecutive traces with a **Stride Predictor** [20].

Although a value may not have been seen before, stride prediction can successfully predict it based on previous values if they follow a simple pattern that the stride predictor can identify, such as:

```
1, 2, 3, 4, 5, ...
```

Or

```
1, 3, 5, 7, 9, ...
```

In these two cases, there is a fixed difference between two instances of the same value. The first sequence has a stride of 1 and the second has a stride of 2.

Figure 4 shows an example of a stride predictor. The last value seen is stored with a stride (between two consecutive values). A new input value is predicted by adding the stride and last value seen. This extension allows the prediction of values in loops, for example.

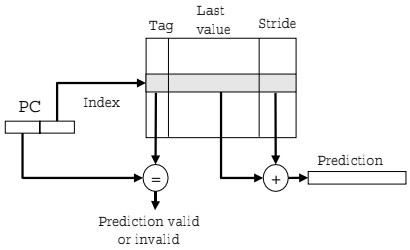

Figure 4. Stride predictor

To implement stride recognition, RST must compare two consecutive input and output contexts of a trace to check for a constant stride. After verifying that there is a possible stride, the stride value must be kept and compared with the next instance of the same trace to see whether the stride remains constant in the next iteration.

3.3. Trace creation in Stride-Aware RST

Stride-aware RST requires an enhanced trace construction mechanism. The last created trace is kept in a temporary buffer in the RS4 stage (see Figure 1), where the trace

PC address, inputs and outputs are compared to the next created trace. If the input and output register identifiers are the same and the values are different, then the difference (i.e., the stride) is stored. The input and output contexts for the temporary trace in RS4 are also updated with the new value. If the next created trace has the same stride when compared to the previously stored trace, then the trace in *Memo_Table_T* is updated with the stride information (the stride and the last seen value for each input). A trace input may now be predicted from its stride and last value when execution reaches the address of the first instruction belonging to the trace. When this trace is accessed and reused, the last value for the trace in *Memo_Table_T* is updated with the new value. Traces with branches are not considered due to the possible dependencies among predicted trace inputs and branches, which would complicate the hardware for speculative trace reuse. Otherwise, stride-aware RST works in the same manner as speculative reuse in regular RST.

Figure 5 compares *Memo_Table_T* entries for RST and stride-aware RST. Fields that are specific to stride prediction are marked in gray. Other than the register identifiers and values (*cr* and *cv*) for the input and output contexts, stride-aware RST also needs fields (*cd*) for the trace stride and a counter (*it*) for the maximum number of iterations seen in the last time that the trace was speculatively reused. The values for the current iteration can be saved in a separate structure to save space in *Memo_Table_T* .

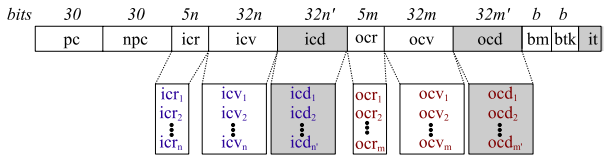

Figure 5. *Memo_Table_T* entries for stride-aware RST

A trace may have as many strides in the input context as the number of predicted values for each speculative reuse. The same is not true for the output context, as the variation of each input value may affect more than one output. However, the number of outputs is usually smaller than the number of inputs for traces, and the number of outputs can be arbitrarily limited to avoid overly enlarging *Memo_Table_T* entries. Traces with strides do not need the branch mask fields, however these are limited to a few bits and do not make much difference in terms of implementation costs.

3.4. Implementation issues

Traces with a stride pattern in their input contexts are hard to predict when last n-value predictors are used. The last correctly predicted trace for a given PC is not likely to be the correct one in the next prediction. Although simple confidence mechanisms based on counters provide acceptable misprediction rates for the cases where last value prediction suffices, they do not account for the effect of having a stride between correct predictions. Therefore, a separate strategy to deal with stride predictions may improve prediction rates.

After a sequence of traces with a stride pattern is identified by RST, there are two strategies to be pursued after marking the original trace: *(i)* avoid further predictions as the trace is not predictable by the last n-value predictor or *(ii)* allow stride predictions.

The first strategy is simpler and does not require extra fields in *Memo_Table_T* , but it does not take advantage of recognizing strides other than to avoid predicting them. On the other hand, the second strategy may increase speculative trace reuse opportunities and performance. It does increase hardware complexity to hold the stride for input and output values.

Both strategies must recognize strides between two consecutive traces. This hardware consists of an extra set of buffers for the input and output contexts, as well as adders and comparators. Extra registers to keep the difference between two trace iterations are also required. This extra hardware does not affect the main pipeline and, as the operations are very simple (adding and comparing values), they do not increase the overhead in the RST mechanism.

4. Results

In this section, we present and discuss our experimental results for RST with and without stride prediction. We first describe our methodology and then experimental results.

4.1. Simulation Environment

Our simulation environment used the *sim-outorder* superscalar simulator from the SimpleScalar Tool Set [1], version 3.0b. The simulator was modified to model a detailed superscalar processor that can be configured with a range of pipeline depths. Our experiments were done with a 19-stage configuration to simulate a up-to-date microprocessor, depicted in Figure 6.

Three different implementations of RST were used in the experiments: *(i)* n-value RST that uses last n-value prediction to speculate trace inputs; *(ii)* stride-aware RST that uses n-value and stride prediction; and *(iii)* filtered-stride RST that uses n-value prediction and stride identification. This

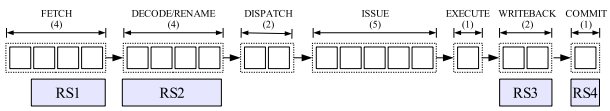

Figure 6. Pipeline configuration (19 stages)

last implementation uses the stride recognition to avoid making predictions on trace inputs that present strides and possibly low redundancy. Filtered-stride RST is used in the experiments to measure how well the n-value RST reuses strided traces, although it has no knowledge of strides. That is, the original RST may capture many of the same traces as stride-aware RST, particularly in inner loops when traces are re-accessed frequently.

The architecture configuration for all experiments is summarized in Table 1. This configuration is used because it has shown to have good performance [17]. In all results, RST is allowed to predict at most two trace inputs.

Parameter	Value
Pipeline width	4
IFQ	16 instructions
RUU	128 entries
LSQ	64 entries
Branch predictor	gshare
First level	13-bit reg (XORed with PC)
Second level	8192 entries
BTB	4096 entries, 2-assoc
Memo_Table_T size	1024 entries, 4-assoc
Memo_Table_G size	2048 entries, 4-assoc
Squash table	64 entries
Confidence mechanism	saturated counters
First level	4096 entries
Saturation	3
Threshold to predict	3
Penalty	3
Correct prediction inc	1
Initial value	1
First level caches	32 KB each, 1 cycle
Second level cache	512 KB each, 5 cycles
Third level cache	2 MB, 10 cycles
Memory	100 cycles
Memory access width	16 bytes

Table 1. Architecture configuration

The workload is composed of programs and input sets from SPEC CPU 95int and 2000int [23]. Integer benchmarks were chosen because RST does not reuse floating-point instructions. These benchmarks were compiled with GNU gcc 2.7.2.3 for PISA (from the University of Michigan), with the flags -O3 -funroll-loops. When available, reduced input sets [10] were used for SPEC2000 benchmarks. All benchmarks were executed for 500 million committed instructions or until completion.

We start the experimental results with the difference in the number of reused traces in Section 4.2. Next, we show how stride traces can change the distribution of committed instructions that are bypassed by trace reuse, speculative trace reuse or instruction reuse in Section 4.3. Finally, we present speedups for the different RST implementations in Section 4.4.

4.2. Number of reused traces

Figure 7 shows the number of reused traces for the different value predictors. For each benchmark, the three bars show results for n-value RST, stride-aware RST, and filtered-stride RST. Each bar shows the number of traces that are regularly reused without prediction, reused with prediction, and misspeculated. The last set of bars is the average for all benchmarks.

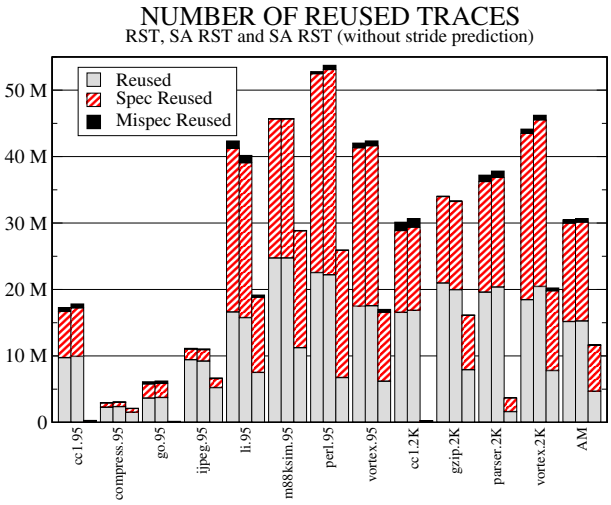

Figure 7. Number of reused traces for n-value RST and stride-aware RST

Stride recognition increases the number of reused traces by less than 1%. However, the number of misspeculated traces also grows with stride prediction, which may mitigate any performance gain from the small increase in the number of reuse traces. On the other hand, if traces with strides are not allowed, then the number of reused traces decreases by almost 40%. From this result, it is clear that capturing stride is important to the number of reused traces and

the n-value predictor can capture strides, despite having less knowledge about the behavior of trace inputs.

4.3. Contribution of reused instructions

Figure 8 shows the effect of prediction on the distribution of committed instructions. For each benchmark, there are two bars that correspond to n-value RST and filtered-stride RST. For clarity, we do not show results for stride-aware RST as the results are similar to RST with n-value prediction. The bars show the percentage of instructions committed due to instruction reuse, trace reuse without speculation, trace reuse with speculation (i.e., at least one input was predicted), and no reuse of any sort.

As the figure shows, a significant percentage of instructions are committed due to speculative trace reuse with last n-value prediction (the first bar for each benchmark). However, when traces with strides are avoided (the second bar for each benchmark), there is a drop in the number of committed instructions due to trace reuse. For example, in *cc1* and *go* almost all the reuse opportunities are due to instruction reuse. Indeed, avoiding traces with strides increases the amount of instruction reuse in all benchmarks, which indicates the presence of redundancy and reuse potential that is not captured by filtered-stride RST.

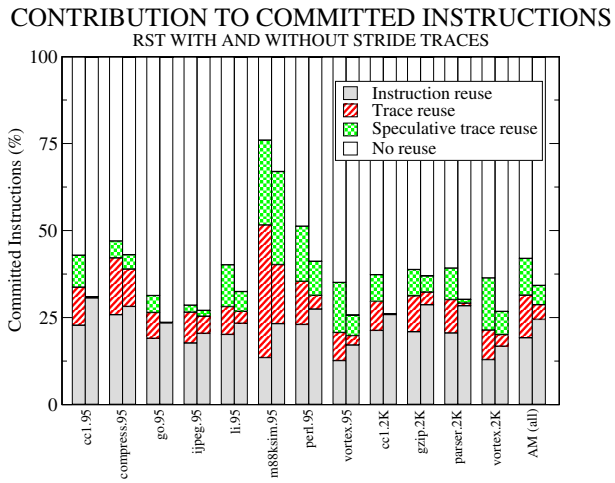

Figure 8. Contribution to committed instructions for n-value prediction and filtered-stride-value prediction

Because n-value and filtered-stride RST both use n-value prediction, the drop in the number of reused traces with filtered-stride RST is due to stride values. That is, the n-value predictor actually captures a large number of trace input contexts that have stride values. This behavior is partly due to inner loops, where previous trace inputs are needed again on future iterations of the outer loop. Although the stride-aware predictor can capture these values earlier than the n-value predictor (i.e., it predicts stride values without previously seeing the trace), this "early capture" does not have a significant advantage when the same trace is re-executed many times.

4.4. Speedup

Figure 9 shows the speedup for n-value RST, stride-aware RST, and filtered-stride RST over a baseline architecture without trace reuse. The last set of bars shows the harmonic mean improvement for all benchmarks. As the figure shows, n-value RST and stride-aware RST improve performance significantly, with a harmonic mean speedup of 1.30. The n-value and stride-aware predictors lead to the same performance and the small gain from reusing more traces with stride-aware RST is lost when performance is considered. Filtered-stride RST has much lower performance than n-value RST, with a harmonic mean speedup of 1.24. For some benchmarks, such as *cc1.95*, *go.95*, *vortex.95*, *cc1.2k*, *parser.2k*, and *vortex.2k*, different traces starting at the same instruction address differ only in the values in an input context are very important to performance. Filtered-stride RST does not reuse these trace, which causes performance to drop by 50%.

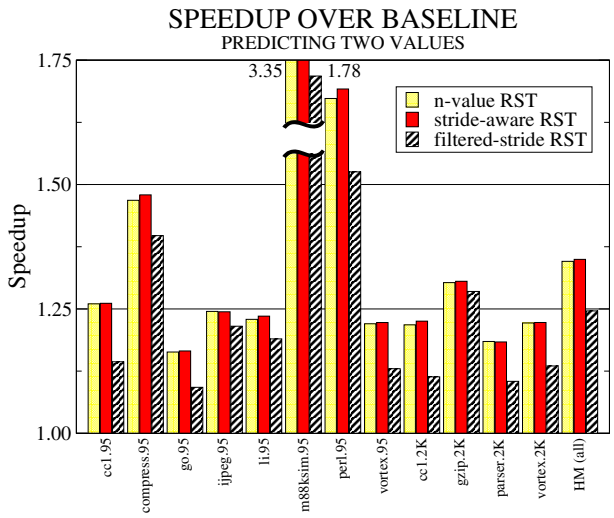

Figure 9. Speedup over baseline for RST and stride-aware RST

From these results, we can make two conclusions. First, stride values are important to achieving good performance from RST. Second, n-value prediction works nearly as well

as stride-aware prediction, yet it is much simpler to implement and requires less knowledge about value behavior than stride-aware prediction. The performance of stride-aware prediction might be partly due to our predictor's limitation that branches are not allowed inside of traces with stride predictions. In future work, we will investigate the impact of this restriction and whether the complexity of allowing branches inside strided traces is worth a performance gain.

5. Previous Work

The concept of *Value Reuse* in computer architectures has come from the observation that many instructions are executed with exactly the same inputs, generating the same results [11, 20, 22].

Value reuse can improve performance in four ways [21]: instructions reused are not executed, which may save cycles for instructions with high latencies; results are ready earlier, allowing dependent instructions to start execution earlier; useful work in wrong paths may be preserved; and value reuse collapses data dependencies, and dependent instructions may be executed in parallel. Additionally, some mechanisms may improve branch prediction by correcting mispredictions when instructions are reused [3] and also use the redundancy of instructions in diverse locations that execute the same computations [12].

Many different mechanisms based on value reuse have been proposed. The unit size to be reused may be instructions [18, 22], expressions and invariants [12], basic blocks [9], traces [3, 5], as well as instruction blocks and sub-blocks of arbitrary size [8]. Some techniques require compiler assistance in order to achieve reuse [7, 8, 27].

Loads and stores are not usually reused because of side effects and aliasing problems. One approach to implement load and store reuse is to manage registers as a level in the memory hierarchy [29]. Another approach uses instruction reuse to exploit both same instruction and different instruction redundancy [28].

Value Prediction, on the other hand, predicts a value based on its history. Predictable value sequences may be divided in three types [20]: *constant*, where the same value occurs again; *stride*, where there is a stride between two subsequent values; and *non-stride*, where there is a complex correlation or no correlation between two subsequent values.

Many variations on value prediction have been proposed, such as two-level value prediction [26], hybrid value prediction [19, 26], and others, many of which were inspired by previous works about branch prediction. Load value prediction [11] is a specialization where only loads are predicted, motivated by their long latencies and high value locality. Another version [24] uses the value stored in a register as the prediction, without requiring extra tables. Some approaches [14, 27] uses an extra execution engine to deal with mispredictions, increasing the complexity of hardware.

Value prediction based on correlation [13, 20, 26] uses global information about the path to select predictions. Prediction of multiple values for traces of instructions [19] may be done for only the last updated values in a trace, reducing necessary bandwidth in the predictor. Another work [25] suggests finding and predicting chains of dependent instructions in the critical path to optimize performance.

Stride value prediction [20] presents better results than last-value prediction in many cases, because it can extrapolate already seen values to predict their next instances, as it is the case where loops are involved.

Speculation control [6] may be used as a mechanism to balance the benefits of speculation against other possibilities. Thus, confidence mechanisms [2] may be employed to improve value prediction by restricting prediction of unpredictable values.

6. Conclusion

This paper presented a study of two value predictors for RST: last n-value and stride prediction. From our results, we found that stride values are important to getting good performance from speculative trace reuse. However, we also found that last n-value prediction is able to capture stride values when traces are re-accessed. Although stride-aware RST did have a small performance advantage, the extra hardware needed to implement stride identification and prediction is not worth the benefit.

In future work, we intend to test other stride prediction mechanisms with more architecture and confidence configurations. Our current stride-aware prediction strategy does not allow branches inside traces, which may limit the gain from predicting strides. An interesting future direction is to remove this restriction; however, the hardware cost to deal with branch misspeculations inside a trace may be complex. We also intend to investigate using the compiler to annotate code with hints to guide the confidence and trace creation mechanisms to improve the quality of traces and predictions.

References

[1] D. C. Burger and T. M. Austin. The Simplescalar Tool Set, version 2.0. Tech Report CS-TR-1997-1342, Unniv. of Wisconsin–Madison, 1997.

[2] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *Proc. of the 26th Annual Intl. Symp. on Computer Architecture*, p. 64–74, Atlanta, May 1999, ACM.

[3] A. T. da Costa, F. M. G. França, and E. M. Chaves Filho. The dynamic trace memoization reuse technique. In *Proc. of the 9th Intl. Conf. on Parallel Architectures and Compilation*

Techniques, p. 92–99, Philadelphia, Oct. 2000, IEEE Computer Society.

[4] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. Tech Report EE Dept. #1080, Technion–Israel Institute of Technology, 1996.

[5] A. González, J. Tubella, and C. Molina. Trace-level reuse. In *Proc. of the 28th Intl. Conf. on Parallel Processing*, p. 30–37, Aizu-Wakamatsu, 1999, IEEE Computer Society.

[6] D. Grunwald, A. Klauser, S. Manner, and A. Plezskun. Confidence estimation for speculation control. In *Proc. of the 25th Annual Intl. Symp. on Computer Architecture*, p. 122–131, Barcelona, June 1998, ACM.

[7] J. Huang and D. J. Lilja. Exploiting basic block value locality with block reuse. In *Proc. of the 5th Intl. Symp. on High Performance Computer Architectures*, p. 106–114, Orlando, Jan. 1999, IEEE CS.

[8] J. Huang and D. J. Lilja. Exploring sub-block value reuse for superscalar processors. In *Proc. of the 9th Intl. Conf. on Parallel Architectures and Compilation Techniques*, p. 100–110, Philadelphia, Oct. 2000, IEEE CS.

[9] J. Huang and D. J. Lilja. Extending value reuse to basic blocks with compiler support. *IEEE Transactions on Computers*, 49(4):331–347, Apr. 2000.

[10] A. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-base computer architecture research. In *Proc. of the Workshop for Workload Characterization – Intl. Conf. in Computer Design*, p. 83–100, Austin, USA, 2000, IEEE CS.

[11] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proc. of the 29th Annual Intl. Symp. on Microarchitecture*, p. 226–237, Paris, Dec. 1996, IEEE Computer Society.

[12] C. Molina, A. González, and J. Tubella. Dynamic removal of redundant computations. In *Proc. of the 13th ACM Intl. Conf. on Supercomputing*, p. 474–481, Rhodes, 1999, ACM.

[13] T. Nakra, R. Gupta, and M. Soffa. Global context based value prediction. In *Proc. of the 5th Intl. Symp. on High Performance Computer Architectures*, p. 4–12, Orlando, Jan. 1999, IEEE Computer Society.

[14] T. Nakra, R. Gupta, and M. L. Soffa. Value prediction in VLIW machines. In *Proc. of the 26th Annual Intl. Symp. on Computer Architecture*, Atlanta, May 1999, ACM.

[15] M. L. Pilla, P. O. A. Navaux, A. T. da Costa, and F. M. G. França. Predicting trace inputs with dynamic trace memoization: Determining speedup upper bounds. *IEEE TCCA Newsletter*, Oct. 2001.

[16] M. L. Pilla, P. O. A. Navaux, F. M. G. França, A. T. da Costa, B. R. Childers, and M. L. Soffa. The limits of speculative trace reuse on deeply pipelined processors. In *Proc. of the 15th Symp. on Computer Architectures and High Performance Computing*, p. 36–44, São Paulo, Oct. 2003, SBC.

[17] M. L. Pilla, P. O. A. Navaux, F. M. G. França, A. T. da Costa, and B. R. Childers. Reuse through Speculation on Traces for Deeply Pipelined Superscalar Processors. Technical Report 345, II–UFRGS, 2004.

[18] A. Roth and G. S. Sohi. Register integration: A simple and efficient implementation of squash re-use. In *Proc. of the 33rd Annual Intl. Symp. on Microarchitecture*, p. 223–234, Monterey, 2000, IEEE Computer Society.

[19] R. Sathe, K. Wang, and M. Franklin. Techniques for performing highly accurate data value prediction. *Microprocessors and Microsystems*, 22(6):303–313, Nov. 1998.

[20] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proc. of the 30th Annual Intl. Symp. on Microarchitecture*, p. 248–258. Los Alamitos, IEEE CS, Dec. 1997.

[21] A. Sodani. *Dynamic Instruction Reuse*. PhD thesis, University of Wisconsin, Madison, Mar. 2000.

[22] A. Sodani and G. S. Sohi. Understanding the differences between value prediction and instruction reuse. In *Proc. of the 31st Annual Intl. Symp. on Microarchitecture*, p. 205–215, 1998, IEEE CS.

[23] Standard Performance Evaluation Corporation. http://www.spec.org.

[24] D. M. Tullsen and J. S. Seng. Storageless value prediction using prior register values. In *Proc. of the 26th Annual Intl. Symp. on Computer Architecture*, p. 270–281, Atlanta, May 1999, ACM.

[25] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Proc. of the 7th Intl. Symp. on High Performance Computer Architectures*, p. 185–195, Monterey, Jan. 2001, IEEE CS.

[26] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *Proc. of the 30th Annual Intl. Symp. on Microarchitecture*, p. 281–290, Dec. 1997, IEEE CS.

[27] Y. Wu, D.-Y. Chen, and J. Fang. Better exploration of region-level value locality with integrated computation reuse and value prediction. In *Proc. of the 28th Annual Intl. Symp. on Computer Architecture*, p. 98–108, Göteborg, Sweden, June 2001, ACM.

[28] J. Yang and R. Gupta. Load redundancy removal through instruction reuse. In *Proc. of the 29th Intl. Conf. on Parallel Processing*, p. 61–68, Toronto, Aug. 2000, IEEE Computer Society.

[29] S. Önder and R. Gupta. Load and store reuse using register file contents. In *Proc. of the 15th ACM Intl. Conf. on Supercomputing*, p. 289–302, Sorrento, Italy, 2001, ACM.