The Limits of Speculative Trace Reuse on Deeply Pipelined Processors

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

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

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

Bruce R. Childers, Mary Lou Soffa
Computer Science Dept – Univ. of Pittsburgh, USA
{*childers,soffa*}*@cs.pitt.edu*

Abstract

Trace reuse improves the performance of processors by skipping the execution of sequences of redundant instructions. However, many reusable traces do not have all of their inputs ready by the time the reuse test is done. For these cases, we developed a new technique called Reuse through Speculation on Traces (RST), where trace inputs may be predicted. This paper studies the limits of RST for modern processors with deep pipelines, as well as the effects of constraining resources on performance. We show that our approach reuses more traces than the non-speculative trace reuse technique, with speedups of 43% over a non-speculative trace reuse and 57% when memory accesses are reused.

1. Introduction

Although modern processes devote significant resources to extracting instruction-level parallelism (ILP) from programs, control and data dependencies still remain a barrier to effectively exploiting large amounts of ILP. Indeed, the additional complexity introduced by very wide issues may adversely impact the processor clock rate. What is needed are complexity-effective techniques that can increase performance by overcoming or mitigating the impact of control and data dependencies.

It is known that programs execute a large amount of redundant or predictable computations [1, 6, 10, 13, 14]. Many techniques have been developed to take advantage of redundancy to improve performance by not executing redundant computations. Value reuse is one technique that exploits redundant computations by reusing previously computed values. Once a computation is executed, future executions can check if the inputs match previous inputs, and then the result of the computation can simply be reused without computing it. However, the input values needed by the computation must be available when checking for reuse.

It is also well known that many of the values during program execution can be predicted correctly [6, 10, 13]. By predicting values, the impact of true data dependencies can be mitigated by letting more instructions execute in parallel. Value prediction may also hide high latencies. When value prediction is employed, unlike reuse, the predicted value must be validated by actually computing the value and checking it against the predicted value. On a misprediction, any computation using the predicted value directly or indirectly has to be recomputed.

In this paper, we present a technique, called Reuse through Speculation on Traces (RST), that combines both value reuse and value prediction for instruction traces. The goal of RST is to increase the number of instruction traces that can be reused by predicting the values of trace inputs that are not available when applying reuse. We explore to what extent value prediction and trace reuse can be effectively combined to improve performance. Our work is a limit study that investigates the potential of different RST models for improving performance in deeply pipelined superscalar processors. The study illustrates the effects of several parameters of RST on performance. We show that speculatively executing traces can be an effective technique for improving performance with a simple reuse model and small hardware tables.

In the next section, we describe the RST technique in more detail. In Section 3, we present the goals of our limit study, our trace reuse models, and the architecture configurations used for the study. The results of our limit study are given in Section 4. We also investigate how to constrain the reuse model in Section 5 to get good performance with less resources. Concluding remarks are given in the last section.

Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03)
0-7695-2046-4/03 $17.00 © 2003 **IEEE**

2. Reuse through speculation on traces

The key idea of RST is to speculate some of the input values of a trace if they are not ready, rather than waiting for their computation to end or not applying reuse. Value prediction is done when some of the input trace registers match stored values and other input values are not available. It is these latter values that are predicted. RST is an integrated mechanism that combines trace reuse and value prediction. It is also designed to be a complexity-effective approach using most of the hardware that is already present for trace reuse.

Traditional value reuse is non-speculative. After the input values of a set of instructions are verified against stored values and a match is found, their results can be reused without executing the instructions. Importantly, resources are not wasted due to reuse and are available to other instructions. The main disadvantage is that reuse must wait until all the input values are ready to be tested for reuse. Therefore, many cycles that could be saved by reusing instructions may be spent waiting for input values that were not ready at the time of the reuse test.

On the other hand, value prediction can overcome the limits imposed by true data dependencies [10, 13]. Instructions with true data dependencies may be executed in parallel when value prediction is employed. This technique may also hide latencies of instructions accessing memory or with high complexity. The main disadvantage is that mispredictions can incur a high recovery penalty. Another disadvantage is that, since value prediction increases concurrency and demands for resources, instructions executing with mispredicted values may prevent the execution of useful instructions.

Trace reuse has been proposed to improve performance by not computing redundant sequences of instructions [3]. The three stages of trace reuse are shown in Figure 1. The *reuse domain* is defined as the set of instructions that can be reused and do not present side effects. First, in Figure 1(a) instructions in the reuse domain are identified (gray circles) and stored. In the next execution shown in Figure 1(b), these instructions are marked as redundant and a trace is formed, until an instruction that does not belong to the reuse domain or is not redundant is found (black circle). This trace is memoized and stored in a memoization table. Figure 1(c) shows the next time execution reaches the beginning of this trace with the same inputs, when the memoized trace is reused; i.e., the previous values are written in the output registers. In this example, the input registers compared are *r1*, *r2*, *r3* and *r9* using stored values for these registers. If the inputs match, the values stored for *r5*, *r6*, *r7* and *r9* are loaded into these registers as the outputs of the trace. Thus, all instructions inside the trace are essentially collapsed into the checking of the inputs and storing of the outputs. The instruction fetch is redirected to the next address after the trace. More information about trace construction is available at [3, 4].

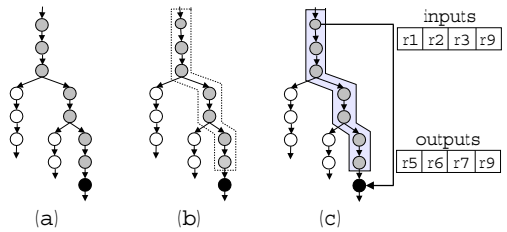

Figure 1. Trace: (a) identification and construction, (b) memoization, (c) reuse

RST combines the advantages of both value prediction and reuse. Unavailable inputs for memoized traces (input and output values stored) are predicted by RST. When traces are reused speculatively in RST, the output values are sent directly to the commit stage, as well as to the instructions waiting for these values and to the register file. Dispatch, issue, and execution are bypassed for the *entire* trace in a single cycle. Therefore, speculative reuse does not increase but reduces the pressure on valuable resources such as functional units.

Applying reuse and value prediction separately but at the same time could require a prohibitive amount of storage in tables. Because we integrate the techniques, RST does not need extra tables to store values to be predicted. The input context of each trace (the input values of all instructions in the trace) already stores the values for the reuse test, which may also be used for prediction. Thus, our proposed technique minimally increases the hardware needed to implement speculative trace reuse, when compared to the hardware needed for non-speculative trace reuse.

RST may reuse both instructions and traces, but only traces are speculatively reused because they encapsulate many instructions and possibly critical paths, thus allowing more performance improvement than single instructions.

Compared with instruction reuse techniques [14], RST has all the benefits of trace reuse, such as the potential for collapsing critical paths into a single cycle, improving branch predictions, and reducing the fetch bandwidth needed. It is also simple to implement as it does not need to involve compiler or ISA modifications such as those needed in block and sub-block reuse [8, 18], allowing the execution of legacy code without modifications. Unlike other trace reuse mechanisms [3, 7], RST can speculatively reuse traces when inputs are not ready. Previous value prediction techniques [6, 10, 13, 17] use more resources when mispeculations occur, while RST is more conservative: predicted

traces are not executed, but speculatively reused.

3. Trace reuse model

In designing an RST mechanism, there are many choices that could be made. For example, the size of the tables that store input values for traces impacts the number of candidate traces for reuse. As another example, it may be beneficial to allow load and store instructions to be part of a trace. However, including memory operations on traces will significantly increase the hardware complexity. In general, the best configuration of the RST mechanism is difficult to determine *a priori* and is influenced by the program workload. In this work, we investigate and answer the questions about the limits of RST and how aggressive the hardware mechanisms must be to achieve significant performance improvements.

We first considered the potential (and unrealistic) performance gains (limits) for RST, using an oracle and an aggressive superscalar architecture, with a superscalar being our base architecture. We then performed experiments to better understand the factors that influence the performance of RST. To explore the performance gains in a more realistic setting, we place restrictions on our architecture and reuse model. In the next sections, we describe our experiments and results for the limit study followed by the experiments and their results when restrictions are placed.

3.1. Trace Reuse without and with Speculation

To investigate the characteristics and potential of RST, we extended the *sim-outorder* microarchitecture simulator from SimpleScalar [2] to support trace reuse without and with speculation. Our extension also models a deeply pipelined wide-issue superscalar processor (see below). For trace reuse without speculation, we use Dynamic Trace Memoization (DTM) [3, 4, 5]. The DTM hardware collects a sequence of redundant instructions across branches as a trace. The traces are stored in two hardware tables called *Memo_Table_G* and *Memo_Table_T* . When a redundant instruction is found in *Memo_Table_G* , DTM starts to build a trace as depicted in the previous section and in [3, 4]. For this study, we considered fully-associative tables.

To implement trace reuse with speculation, we extended DTM to have the capability to speculate unavailable trace input values. The reuse test in DTM only identifies a trace as reusable when all inputs are ready and equal to the values stored in *Memo_Table_T* . RST introduces a modification in the reuse test to allow speculation using a previous value for a missing trace input. If a value in the input context is not ready, but the remaining values match the current values, then a confidence mechanism can be accessed to verify if the value should be predicted or not.

In our experiments on RST, we limit prediction to one register value for each trace, as it is more likely that these traces will be reusable in implementations. In some cases, loads and stores were included in the reuse domain and included in traces. We also use an oracle confidence mechanism for value predictions, which knows when a prediction is correct or incorrect, allowing only correct predictions.

To investigate the limits of RST and the characteristics of traces from RST, we used a number of different architecture and reuse models, including:

- **DTM:** Trace reuse without speculation of unavailable inputs;
- **DTM-M:** DTM with loads and stores in the reuse domain;
- **RST:** Trace reuse with speculation of one unavailable input using oracle confidence;
- **RST-M:** RST with loads and stores in the reuse domain; and
- **RST-R:** Restricted RST-M, where only loads served by stores in the same trace can be reused; a load that is not served by a store inside the trace terminates trace construction.

3.2. Architecture Configuration and Benchmarks

For our study, we use a baseline architecture that is comparable to aggressive superscalar processors. The baseline is augmented with hardware support for trace reuse in the DTM, DTM-M, RST, and RST-M models. The baseline superscalar has a 20-stage, 4-wide pipeline with three cache levels. The execution pipeline has three integer FUs, with one of them capable of doing multiplication and division. The branch predictor is a 2048-entry combined predictor, using Gshare with 1024 entries and 13 bits of history and bimodal with 2048 entries. The return address stack has 32 entries. The IFQ allows 16 instructions, the RUU has 128 entries, and the LSQ allows 64 loads and stores to be simultaneously in flight.

The i-l1 cache is 2-way set-associative 16 KB and the d-l1 cache is 4-way set-associative 16 KB. The latency of a hit in the first level is two cycles. The remaining caches were unified. The l2 cache is 8-way set-associative 512 KB, with a hit latency of 7 cycles. The l3 cache is 8-way set-associative 2 MB; the hit latency is 14 cycles. Finally, the memory access latency is 200 cycles. The memory width is two words.

For the initial set of experiments with RST and DTM, we used *Memo_Table_G* and *Memo_Table_T* table sizes of 16 K entries. Such large tables capture the majority of traces in a program.

A limitation imposed by our simulation environment was that 64 instructions is the maximum number of instructions

in a trace. However, this limitation has a negligible effect since the vast majority of traces had lengths of less than 64.

The benchmarks chosen for this work are a subset of SPEC95int and SPEC2Kint [15]. We simulated *compress95, ijpeg, li, m88ksim,* and *perl* from SPEC95, and *art, cc1,* and *vortex* from SPEC2K. Only integer benchmarks were chosen because of the intrinsic difficulty of reusing and predicting floating point values [4]. Our benchmarks were compiled with GNU gcc 2.7.2.3 for SimpleScalar PISA, with the flags -O3 -funroll-loops. When available, reduced input sets [9] were used for SPEC2000 benchmarks. Each simulation was executed with a maximum of 1 billion committed instructions, or until completion.

4. Limit study of RST

For the limit study, we explored different two different reuse domains: trace reuse without loads and trace reuse with loads and stores. Our first simulations did not constrain the number of input and output contexts or table size. In a later section, we look at the impact of reducing the hardware structures on RST's performance.

4.1. Impact of two reuse domains

In the first set of experiments, the goal was to discover the performance gains possible with RST in an aggressive superscalar machine. The results are given in Figure 2, which shows the potential performance gain over the superscalar baseline. The last set of bars is the harmonic mean across all benchmarks. In general, the speedups of RST over the baseline is 70% and the speedup of RST-M is 85.3%. Most benchmarks have improved performance when reusing loads and stores, but some benchmarks are particularly effected by memory reuse, such as *m88ksim*. For this benchmark, RST-M presents a speedup of 115% over RST.

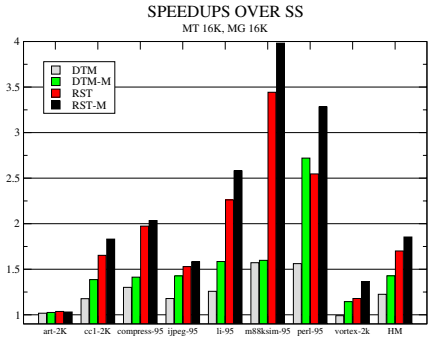

Figure 2. Speedup over baseline architecture

m88ksim has a large amount of redundancy, and the speedups for both reuse domains (i.e., with and without memory operations) are greater than all of the other benchmarks. In this case, DTM-M has a speedup of 1.57 times over the baseline and RST-M has a speedup of 3.98 times over the baseline. Interestingly, for *m88ksim*, the benefit of DTM-M over DTM was not significant, with a performance improvement of only 1.6% of DTM-M over DTM. However, the speedup of RST vs. RST-M is much more significant with a performance improvement of 15.6% .

art has a different behavior where trace reuse has little benefit. In this case, the maximum performance improvement is only 3.7%. Indeed, RST-M reduces performance slightly for *art*. The reason performance is little improved by trace reuse for *art* is the high miss rate in the data caches (the first level cache has a 30% miss rate and the second level cache has a 31% miss rate), so memory performance dominates in this case. Because this benchmark is memory bound, a small number of instructions is actually skipped by applying trace reuse. Only 21.6% instructions are skipped for the best case for *art*, while the harmonic mean across all benchmarks is 63.7% instructions skipped. The small incidence of branches (13% of instructions) is also a factor in minimizing the performance gain of RST and DTM for *art*. Reuse can significantly reduce branch mispredictions, and thus applications with more branches are more likely to present better performance improvements.

While *m88ksim* and *art* are outlying cases, performance across the other benchmarks is improved, with a speedup ranging from 1.2 to 2.5 for RST and from 1.3 to 3.25 for RST-M. It is clear from these speedups that RST offers much performance potential in many benchmarks.

Although RST has performance benefits, another question is whether the performance improvement is coming primarily from trace reuse or from the combination of speculation and trace reuse. Our experiments showed that many traces that could be reused were not because some of their input operands were not ready to be compared when the traces were scheduled to execute, which would suggest speculation is important. Figure 3 shows the percentage of traces for DTM that are reused or not, because some of the trace inputs are unavailable by the time the reuse test is done [12]. Thus, a significant number of traces with correct input contexts (average of 68%) were not reused. This situation occurs because trace reuse is very conservative and does not allow reuse until all the trace inputs are known and match previously computed values.

We also compared our RST and DTM performance results to compute the performance improvement of RST over DTM. Figure 4 shows the speedups over DTM. Without memory reuse, RST has a speedup of 1.43 times over DTM. When loads and stores are reused, the speedup is 1.57 times for RST-M. From the figure, it is clear that including spec-

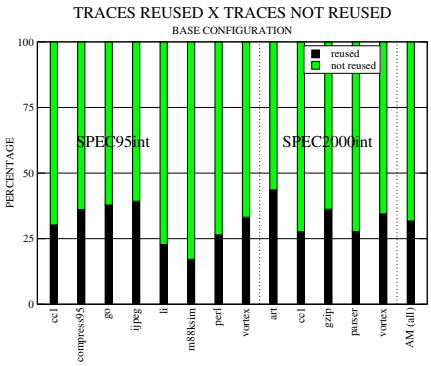

Figure 3. Comparison between traces reused and not reused

ulation to trace reuse is important.

Figure 4 also shows the performance improvement of DTM-M over DTM. In this case, DTM-M improves performance by 17.5% over DTM. However, in most cases, RST has a better speedup than DTM-M. This result indicates that speculation is more important than adding memory operations to the reuse domain. Also, including memory operations as part of the reuse domain is likely to be more complex than including speculation of trace input values. Hence, RST may be a better choice from both a performance and implementation complexity perspective. RST has the disadvantage that it requires a confidence mechanism to mitigate the impact of value mispredictions and this mechanism adds cost and complexity. Later in the paper, we discuss the degree of confidence that is needed to get good performance from RST.

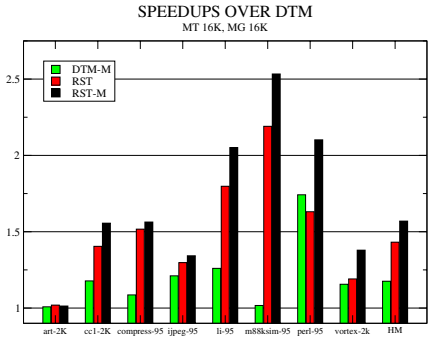

Figure 4. Speedup over DTM

While the figure shows that adding speculation and memory reuse helps, the amount of improvement varies for the benchmarks. For example, in the case of *m88ksim*, adding memory reuse to DTM does not significantly improve performance (1.5%). But adding speculative reuse (RST) improves performance by 119% and RST-M improves performance by 153.3%. For this benchmark, predicting trace inputs is more important than adding memory reuse and memory reuse only shows its real potential when combined with speculation. Likewise, *compress* gets little improvement from memory reuse, but is helped by speculation. *vortex* gets some improvement from memory reuse, but is really helped by combining memory reuse and speculation. From these results, we can infer two conclusions. First, as stated above, speculation of trace inputs is generally beneficial for most benchmarks. Second, the impact of adding memory reuse is mixed and varies based on the importance and ability to reuse memory operations for individual benchmarks. A purely unrestricted form of memory reuse may not be worth its likely complexity with such varied results. Instead, a limited form of memory reuse with less complexity may be worthwhile. We explore such a form of memory reuse later in this paper.

4.2. Factors involved in reuse

We next wanted to understand what are the factors that influence the performance gains of speculative reuse. For example, when we introduce speculation, are there more instructions skipped and what happens to trace size? Does the performance improve because there are smaller traces that are reused more often or is it the case that longer traces are effectively created and reused?

We first experimented with the number of skipped instructions to investigate why performance is improved with trace reuse. Figure 5 shows the percent of reused (skipped) instructions in each benchmark, with the last column being the harmonic mean of percentages. The skipped instructions include instructions reused in traces or individually reused. For most benchmarks, reusing loads and stores increased the percent of skipped instructions, but not for all cases. In *compress95*, there were fewer instructions skipped in RST-M than RST, but the performance of RST-M is better than RST (see *compress95* in Figure 2). This improvement can be explained by the fact that fewer but more expensive instructions are being skipped in RST-M.

perl is another interesting case. Using RST-M, *perl* has a higher percentage of instructions skipped than *m88ksim*, but the performance improvement of *m88ksim* is better than *perl*. This is due to the fact that *m88ksim* skips a lot of loads and stores, which have high latency. The benchmark *vortex* is also interesting. For DTM, *vortex* had a slight performance degradation (about 1%), yet 25% of instructions were skipped in this benchmark. In this case, the traces are relatively small and the overhead associated with trace reuse

to redirect the fetch unit (when applying reuse) overwhelm the benefit of reusing instructions.

The conclusion to be drawn from these experiments is that the percentage of improvement does not simply correlate to the number of skipped instructions but it also depends on the type of instructions that are skipped.

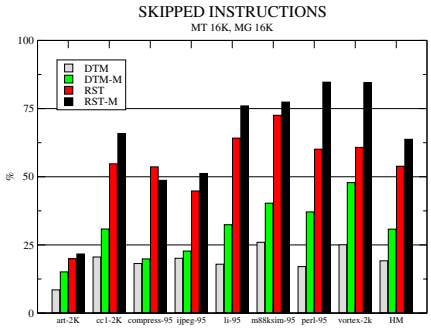

Figure 5. Percent of skipped instructions, unconstrained architecture

We also wanted to explore what happens to the trace length when we add speculation. That is, are we reusing longer traces less frequently or shorter traces more frequently when we add speculation?

Figure 6 shows the average trace lengths for the two versions of DTM and RST. On average, the trace size increases 40% for DTM-M and 27% for RST-M. The difference in trace length from DTM to RST is 15%, and only 4% from DTM-M to RST-M. Therefore, traces are longer in RST than in DTM.

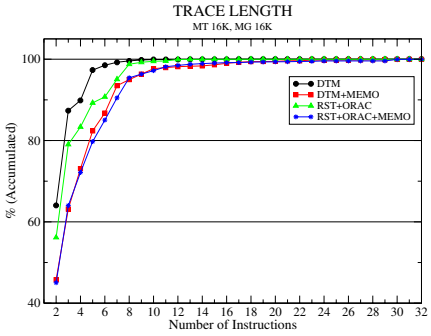

Figure 6. Distribution of trace lengths using baseline

The termination condition is a also a factor in trace length. For DTM and RST, traces are formed dynamically by adding instructions that can be reused (that are redundant). The trace terminates when an instruction is encountered that is not redundant or not part of the reuse domain. Traces may also be terminated by reaching a maximum length of 64 instructions (an artificial restriction imposed by the simulation environment). For this latter case, less than 0.5% of all traces in the benchmarks had 64 instructions, so this restriction plays a negligible part in determining trace length. The other two aspects of trace termination play a more important role in trace length.

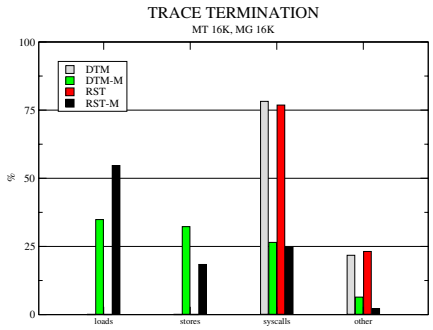

Figure 7. Trace termination conditions

Figure 7 shows the average termination condition for all benchmarks. When memory reuse is not employed, loads and stores finish most traces: about 65% for DTM and 67% for RST. When memory reuse is used, loads and stores are reused when they are redundant and thus become part of the trace. System calls then become the major termination condition. With DTM about 32.7% of all reused traces are finished by system calls and for RST, the number is 26.5%. When memory reuse is employed, the percentages increase to 78.2% for DTM-M and 76.8% for RST-M. Also, the percentage of non-redundant instructions finishing traces (*others* in the graph) increases from 1.9% (DTM) to 21.7% (DTM-M) and from 6.4% (RST) to and 23.1% (RST-M). These results show why including memory operations in the reuse domain can help performance for some benchmarks. Including loads and stores on traces ensures that traces are not terminated too early, which can result in longer traces and better performance. The results also show the importance of skipping redundant memory operations, which can have large execution latencies.

5. Constrained models

We now describe our experimental results that were obtained by constraining some of the parameters to explore a more limited and potentially less complex model of RST.

We studied how the number of inputs and outputs, the loads and stores and the confidence level impacts performance gains.

5.1. Constraining inputs and outputs

The number of trace inputs and outputs, including both registers and memory locations, impacts the size of the underlying reuse tables. We first wanted to know how many memory references are included in a reused trace on average. From this number, we can get an estimate of the size of the contexts for including memory operations as part of the reuse domain.

Figure 8 shows the cumulative distribution of loads and stores when memory reuse is used. Almost all reused traces have at most 3 loads and stores. The average number of loads per reused trace is 1.13 loads for DTM and 0.94 loads for RST. For stores, the average number is 0.37 for DTM and 0.55 for RST. The increase in the number of stores and the decrease in the number of loads reflects a larger redundancy for stores than for loads. Overall, these small averages indicate that including memory operations as part of the reuse domain requires a relatively small amount of storage in the reuse tables.

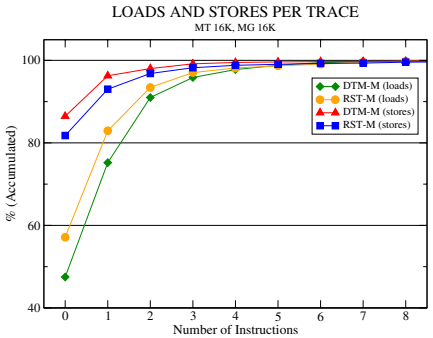

Figure 8. Distribution of number of loads and stores

Because the number of input and output registers for a trace effects the size of the reuse tables, we also investigated how performance changes when the input and output context sizes are restricted. We first found that the maximum input and output context sizes for all benchmarks never went above 16 registers. We then constrained the input and output context sizes to each be 16, 8, and 4 registers.

Figure 9 shows the performance of RST with the constrained input and output context sizes. In the cases of 4 and 8 context sizes, traces were terminated when adding more instructions caused a trace's context size to exceed the limit. From the figure, when the number of inputs and outputs is restricted to 4, there is only a 10% performance degradation in RST's performance (a decrease from an 85% performance improvement to a 68% improvement.) For RST-M, the performance does not change when the size of the input and output contexts are restricted.

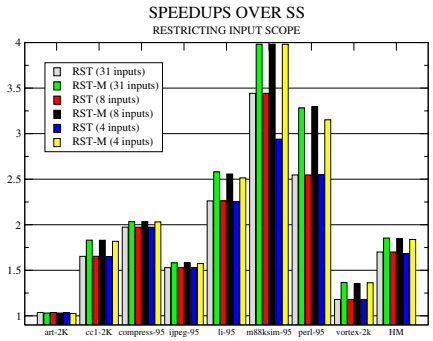

Figure 9. Speedup over baseline with constrained input context size

From the results in figures 8 and 9, we can infer that 4 inputs, 4 outputs and 3 memory values are enough for most traces, providing almost the same performance with a realizable cost. Previous work on DTM [4] showed a similar trend for the number of inputs and outputs.

5.2. Constraining table sizes

In our next experiments, we constrained the size of both the trace and instruction memoization tables (*Memo_Table_T* and *Memo_Table_G*). In the limit study, the size of each table was 16 K and we varied the table size from 128 to 16 K entries. Figure 10 shows speedups when both history tables are constrained to different sizes for RST and Figure 11 shows the same for RST-M.

In both figures, 8 K and 16 K tables have approximately the same performance. There is a slight change in performance when table size is constrained to 4 K entries. RST with 8 K tables is only 4.3% faster than the 4 K configurations and RST-M with 8 K tables is only 5% faster. We also see a similar change in performance when going to 1 K tables.

The figure also shows that different benchmarks favor different table sizes. For example, in the case of RST-M and *perl*, 1 K tables do as well as 8 K tables. However, *li* favors 8 K tables for RST-M. The performance behavior is also very interesting for some benchmarks at very small table sizes. For example, *m88ksim* has better performance with 128 entries than 256 entries for both RST and RST-M. Such

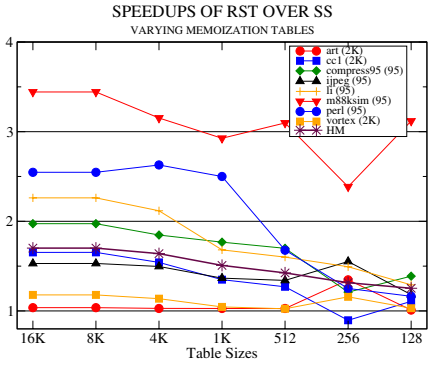

Figure 10. Speedups for RST over baseline with constrained table sizes

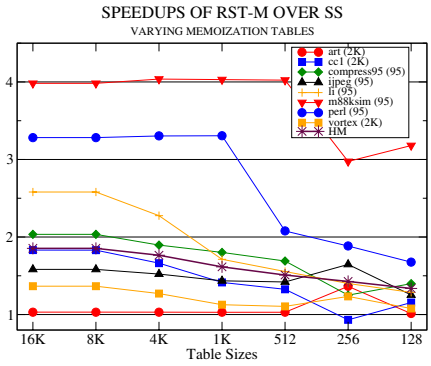

Figure 11. Speedups for RST-M over baseline with constrained table sizes

behavior can be explained by the fact that table size influences what traces actually get captured and are reused. In this case, it happens that more profitable traces are formed with 128 entry tables, rather than 256 entry tables.

Behavior is relatively stable and regular for 512-entry and larger tables. We also varied the size of each table independently and found that a good combination was a relatively small 512-entry *Memo_Table_T* and a 4K-entry *Memo_Table_G* . This combination reduces hardware cost while also achieving good performance, with an average speedup of 1.5 for RST.

5.3. Constraining confidence hit rate

To estimate the impact that the accuracy of the confidence mechanism has on performance, we simulated an adjustable, variable confidence mechanism. The goal of this experiment was to see what level of confidence would be needed to get good performance from trace reuse with speculation. A low level of confidence with good performance would imply that a simple confidence mechanism might be effective at avoiding mispredictions of trace inputs.

We introduced the confidence mechanism as a parameter in our simulations that provided the ability to vary the percentage of accuracy of confidence. We used 100%, 99%, 97%, 95%, 90%, 85%, 80%, 75%, and 70% confidence values. We compared each of the memory models to the same model in non-speculative trace reuse. Such a variable confidence mechanism is useful in determining coarse estimates of the impact of mispredictions on performance, but it is not entirely representative of what would happen in an actual implementation. For this experiment, we assume an uniform distribution of mispredictions and an implementation is not likely to follow such a distribution.

Figure 12 shows the impact of different confidence rates on average performance. The results in the figure are normalized to DTM. The figure shows that at high confidence rates, RST-M has the best performance, followed by RST-R, and then RST. When the confidence rate drops to 99%, there is a drop in performance, as expected. This initial drop is due to the inclusion of mispredictions and the impact of the misprediction penalty.

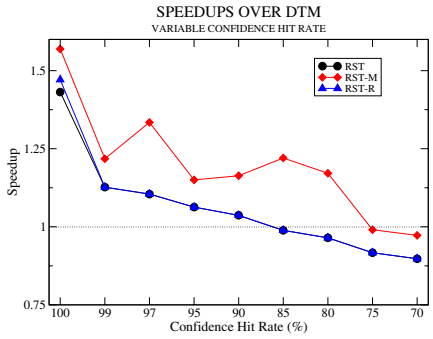

Figure 12. Effect of confidence hit rates on speedups

Across all confidence levels, RST-M has the best performance and continues to show a performance improvement until confidence reaches 75%. The irregular behavior of RST-M can be explained by our simple uniform model for varying the confidence rate. However, the general trend is accurate and reflects that as confidence drops, the impact of mispredictions begins to overwhelm the benefit of trace reuse. RST-R and RST have similar performance at confidence levels of 99% and below. In both cases, there is a performance improvement over DTM until a confidence level of 90%.

The figure shows two important trends. First, to get the most from adding speculation to trace reuse, we need ac-

curate and good confidence mechanisms. Some method to constrain predictions for trace inputs is important to avoid over-speculating. Also, when the confidence mechanism is considered, RST is a better choice than RST-R since the latter does not offer any performance advantages (but adds complexity). However, there is still much potential for memory reuse even when the confidence mechanism is included, as demonstrated by RST-M.

6. Conclusion

This paper explored the potential of adding value prediction and memory reuse to instruction trace reuse in deeply pipelined superscalar processors with a new technique called Reuse through Speculation on Traces (RST). The paper showed that adding speculation to trace reuse improves the amount of reuse and overall performance. It also demonstrated that including memory operations as part of reused traces offers further performance for some benchmarks. We also evaluated the impact of constraining speculative trace reuse with small hardware structures, including the size of the input and output contexts and memoization tables, and limited memory reuse. In particular, our study showed that:

- Adding speculation to trace reuse can improve performance by harmonic mean of approximately 1.7 times over non-speculative trace reuse,
- Including memory operations in the reuse domain with speculation can improve performance by a harmonic mean of 1.85 times,
- 4 input and output registers and 3 load and store contexts are enough for most traces,
- Reuse tables with 512 to 4K entries had good performance and where competitive with much larger tables,
- Confidence levels of 90% to 99% are needed to avoid over-speculating input values.

From this study, we believe that Reuse through Speculation on Traces has much performance potential and our current work is focusing on integrating and developing the hardware structures required for a complexity-effective RST architecture.

Acknowledgments

This work was partially funded by a grant from CNPq Research Agency (Brazil), and by a grant from CAPES Foundation (Brazil). We also thank the LabTec UFRGS/Dell Project and FINEP for the clusters were we simulated the experiments.

References

[1] R. Bodik, R. Gupta, and M. L. Soffa. Load-reuse analysis: Design and evaluation. In *SIGPLAN PLDI*, p. 64–76. ACM, 1999.

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

[3] A. T. da Costa, F. M. G. França, and E. M. Chaves Filho. The dynamic trace memoization reuse technique. In *9th PACT*, p. 92–99, 2000, IEEE CS.

[4] A. T. da Costa. *Exploiting Dynamically the Reuse of Traces in Processor Architecture Level*. PhD Thesis, COPPE–UFRJ, 2001.

[5] L. M. F. de Aquino Viana. Dynamic trace memoization with reuse of memory access instructions, MSc Dissertation, COPPE–UFRJ,2002.

[6] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. TR EE Dept. TR #1080, Technion–Israel Inst. of Technology, Israel, 1996.

[7] A. Gonzalez, J. Tubella, and C. Molina. Trace-level reuse. In *28th ICPP*, p. 30–37, 1999, IEEE CS.

[8] J. Huang and D. J. Lilja. Exploiting basic block value locality with block reuse. In *5th HPCA*, p. 106–114, 1999, IEEE CS.

[9] A. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-base computer architecture research. In *Workshop for Workload Characterization – ICCD*, p. 83–100, 2000, IEEE CS.

[10] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *29th MICRO*, p. 226–237, 1999, IEEE CS.

[11] T. Nakra, R. Gupta, and M. L. Soffa. Value prediction in VLIW machines. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, ACM, 1999.

[12] M. L. Pilla, P. O. A. Navaux, F. M. G. França, A. T. da Costa, B. R. Childers, and M. L. Soffa. Reuse through speculation on traces: preliminary results. TR RP–325, PPGC–UFRGS, 2002.

[13] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th MICRO*, p. 248–258, IEEE CS, 1997.

[14] A. Sodani and G. S. Sohi. Understanding the differences between value prediction and instruction reuse. In *31st MICRO*, p. 205–215, IEEE CS, 1998.

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

[16] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *7th HPCA*, p. 185–195, IEEE CS, 2001.

[17] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th MICRO*, p. 281–290, IEEE CS, 1997.

[18] Y. Wu, D.-Y. Chen, and J. Fang. Better exploration of region-level value locality with integrated computation reuse and value prediction. In *28th ISCA*, p. 98–108, ACM, 2001.