A Speculative Trace Reuse Architecture with Reduced Hardware Requirements

Maurício L. Pilla
Computer Science School–UCPEL
Pelotas, Brazil
pilla@ucpel.tche.br

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

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

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

Philippe O. A. Navaux
Computer Science Inst.–UFRGS
Porto Alegre, Brazil
navaux@inf.ufrgs.br

Abstract

Trace reuse is an effective way of improving the performance of superscalar processors by skipping the execution of a sequence of instructions with known input and output values. However, the extra hardware complexity is of special concern when implementing such mechanisms. In this paper, we describe ways to reduce these requirements for Reuse through Speculation on Traces (RST). RST combines instruction and trace reuse with value prediction in an integrated mechanism to provide missing trace inputs when execution reaches the beginning of a trace. Speculatively reused traces do not consume resources in the execution pipeline, as they are not executed. In this paper, we study the effects of constraining reuse tables to effectively reduce the number of reuse candidates and comparisons. We compare our approach to instruction reuse, trace reuse and value prediction. We show that RST reuses more instructions and has better performance than traditional trace reuse, with an average speedup over a baseline without reuse of 1.21.

1. Introduction

Superscalar processors employed in high-performance computing systems use aggressive speculation to achieve maximum performance. However, control and data dependencies are often significant impediments to obtaining better performance in these processors. Simply increasing available resources may not be the best choice because more hardware complexity can degrade the clock rate. Data dependencies also limit instruction-level parallelism, mitigating gains from using better branch prediction, larger caches, deeper pipelines, and more functional units. Therefore, techniques, like value reuse and value prediction, that can successfully overcome these limits and are not overly complex to implement are necessary to improve performance.

Value reuse allows a computation (e.g., an instruction or a trace) to be skipped when the current input matches a previous input, producing the same output. It can be particularly valuable when applied to instruction traces because many instructions can be skipped at once. Value reuse is restricted by the availability of input values when execution reaches a reusable computation; that is, it can not be applied unless *all* inputs are available. For traces, this limitation is particularly severe because a trace usually has more inputs than a single instruction, and there is a higher likelihood that some input will be unavailable. Value prediction, on the other hand, can speculate computation when input values are unknown. But value prediction can significantly increase pressure on machine resources, limiting its benefit.

Our past papers presented a study of how well RST can work in ideal conditions [14], while this paper describes and evaluates a feasible, constrained RST implementation. RST does both value speculation and trace reuse in a single, integrated mechanism that can be implemented in a superscalar pipeline with minimal complexity, as we show in this paper.

In this work, we evaluate different ways to reduce hardware complexity for a RST implementation: excluding the instruction reuse table, reducing the number of live-in and live-out registers, and reducing table associativity. Our simulations use a baseline architecture that is similar to the IBM Power5 and Intel Pentium 4. We show that our approach has excellent performance, with an average speed of 1.21 over the baseline (harmonic mean). We discuss several possible implementation issues and compare our approach to instruction reuse, trace reuse, and value prediction. The RST mechanism works better than these approaches, yet it has low complexity and can be easily implemented in a superscalar pipeline.

This paper is divided into the following sections. First, the technique of reuse through speculation on traces is presented in Section 2. Section 3 describes an architecture for RST. Section 4 experimentally evaluates RST and compares it to other alternatives. Section 5 discusses related work. Finally, Section 6 presents a summary of the paper.

2. Reuse through Speculation on Traces

RST improves trace reuse and hides true data dependencies by allowing traces to be *(i)* **regularly reused** when all inputs are ready and match previously stored input values or *(ii)* **speculatively reused** when there are unknown trace inputs. Therefore, traces that could not be reused in previous approaches may be reused to exploit their redundancy.

Value reuse is non-speculative. After the input values of an instruction are verified against previous values and a match is found, the instruction can be reused without further execution. Resources are never wasted due to reuse and are available to other instructions. Trace reuse [3, 5] has been proposed as a way to go beyond single instruction reuse. Here, a sequence of instructions–possibly spanning multiple branches–is the granularity of reuse. Input values to a trace are compared against previous values to check for a reuse opportunity, and when the inputs match the previous values, previously recorded outputs are used to update the architecture state. In a single cycle, all instructions inside a reused trace may be skipped and instruction fetch redirected to the address after the trace end.

A shortcoming of trace reuse is that many traces are not reused because some inputs are unavailable when a trace is accessed. Indeed, it can take many cycles after execution reaches the start of a trace before all inputs are available, which limits the benefits of trace reuse. Previous studies [14] show that only half of possible traces are reused in a non-speculative trace reuse architecture due to input values not being ready to be compared with stored traces. However, reusing only those traces allowed for an average speedup of 1.19 over an architecture without reuse.

Traditional value prediction overcomes the limits imposed by true dependencies by executing instructions with true dependencies in parallel [12, 18]. This technique can also hide instructions with long execution latencies. Value prediction does have a cost: a high misprediction rate and recovery penalty can even lead to worse performance. It can also increase pressure on resources, because the original instruction has to be executed to verify a prediction. A misprediction also wastes the resources with useless instructions.

RST addresses the resource demands associated with value prediction. When traces are reused speculatively, the output values are sent directly to the pending instructions and the register file. Dispatch, issue, and execution are bypassed for the *entire* trace in a single cycle and machine resources *are not occupied by any instruction inside of a trace*. Predictions are made for *trace inputs*, rather than outputs as done in traditional value prediction. Trace inputs are verified by checking predictions as actual values become available during execution. Because inputs are predicted and outputs are supplied by value reuse, a trace does not have to be executed to verify a prediction.

RST does not need extra tables to store values to be predicted because it uses the input values (or "input contexts") already stored in the reuse tables needed for trace reuse. An RST architecture, as we show in this paper, can also be organized in a way to unify the tables normally used for trace reuse to reduce complexity. RST can use the same values needed for non-speculative trace reuse to make "last value" predictions [4]. Because RST uses existing information needed by non-speculative trace reuse, it incurs only a small additional hardware cost.

3. Implementation in a Superscalar Processor

RST has several requirements for implementation in a superscalar processor. An RST architecture needs *(i)* a mechanism to form and store redundant traces; *(ii)* a reuse test to detect reuse opportunities, even when some input values are not ready; *(iii)* a method to reuse a trace and update the architecture state; *(iv)* a mechanism to detect misspeculations; and *(v)* a recovery mechanism for misspeculations. In this section, we describe an RST organization that addresses these requirements.

Traces are created during run-time based on sequences of instructions in a *reuse domain*, which is the set of instruction classes that are allowed to be reused (e.g., integer operations, branches, and address calculations for loads and stores). Each trace has an input and an output context to store live-in and live-out values. The reuse domain is constituted by integer instructions without side effects. Floating-point instructions are not reused because they show little redundancy [3] and because they would require larger registers to be stored in the reuse table. The length of traces may be limited by the number of live-in and live-out values that can be stored, by the number of branches, and by the occurrence of instructions outside of the reuse domain. In loads and stores, the address calculation is split from the actual memory access, and the address calculation can be reused. System calls are not reused because they require a mode change in the processor status. Details on trace construction and reuse can be found in [14].

The RST architecture is organized as a pipeline parallel to the execution pipeline, as shown in Figure 1. The main difference from Figure 1 to previous organizations is that stage RS1 has a single reuse table that holds both instructions and traces. Thus, the number of wires that go from RS1 to RS2 is limited, and the logic in RS4 is simpli-

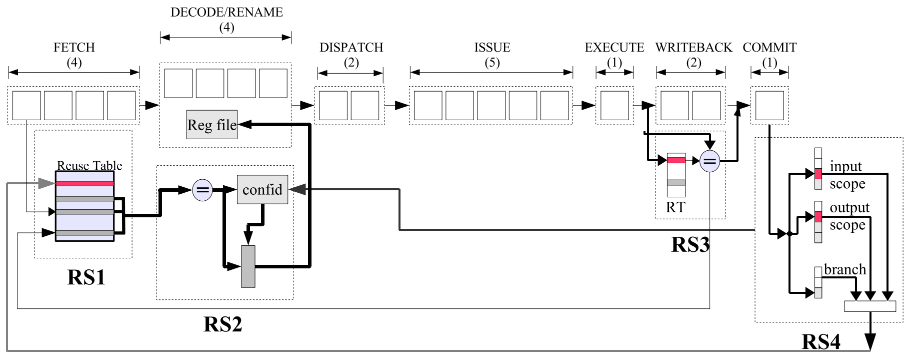
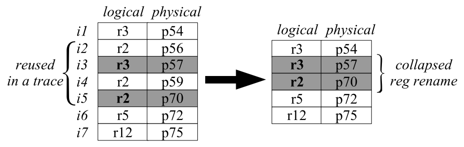


Figure 1. Details of RST's pipeline with unified reuse table


reused
in a trace
collapsed
reg rename
(a)
(b)

(a)

	logical	physical
i1	r3	p54
i2	r2	p56
i3	**r3**	**p57**
i4	r2	p59
i5	**r2**	**p70**
i6	r5	p72
i7	r12	p75

(b)

logical	physical
r3	p54
r3	p57
r2	p70
r5	p72
r12	p75

Figure 2. Discovering register mappings to be freed

fied, which has only to identify traces. Thus, the number of wires from RS4 to RS1 is also reduced, simplifying the design of the processor. In this pipeline, RS1 overlaps the fetch stages, RS2 overlaps the decode and rename stages, and RS3 overlaps writeback. The RST stages also need to communicate with the execution pipeline to access the register file, to get predicted values, and to get committed values and instructions. RS1 is the same initial step done in DTM. RS2 and RS3 are added for value prediction, and RS4 is extended from DTM to handle mispredictions.

Stage RS1 fetches reuse candidates from the reuse table, using the current program counter to address it. Entries in the reuse table are then sent to RS2. The maximum number of candidates sent to RS2 depends on the associativity of the tables. For example, an implementation with two-way set associative tables could send up to two instructions and two traces as reuse candidates for a program counter address. RS1 also receives trace candidates from RS4.

Stage RS2 checks candidates from RS1 for reuse by comparing values from a trace's input context against actual values in the register file. When all inputs match register values, the trace is reused without speculation. However, when an input is unavailable, RS2 predicts missing inputs using values from the reuse table. We use a confidence scheme based on a table of saturating counters [2].

When a trace is reused, the output values are sent directly to dependent instructions. This presents a challenge because by the time that RS2 can decide if a trace is going to be reused, the tables that map logical registers onto physical registers and keep the active list will be in an inconsistent state. For example, consider the register mapping in Figure 2(a). The list would be analyzed from the last instruction in the trace to the first instruction in the trace. Logical registers *r2* and *r3* are written by the trace. Thus, we must search for the last write for each of these registers in the list. In Figure 2(b) instructions with the last mapping to the registers in the output context are marked in gray.

Here, the processor knows that it can free the mappings for registers *p56* and *p59*, and that *p57* and *p70* provide the last values inside the trace. After the unnecessary mappings are freed, the mapping is as Figure 2(b). The mappings from reused instructions are freed, except for the last assignments, which are used to write the output scope of the reused trace.

Stage RS3 verifies predictions of a speculatively reused trace. If a misprediction is found, then fetch is redirected to the beginning of the trace. When all predictions are correct, RS3 commits the trace. The prediction outcome is sent to RS2 to update the confidence table entry for that trace.

Because multiple inputs can be predicted for a trace, the Recovery Table (RT) tracks the number of correctly predicted trace inputs. When all inputs are correct, the trace can be committed. On a misprediction, all entries that are related to traces predicted after the misprediction are squashed. The RT is organized similarly to a reorder buffer, and on a misprediction, it is searched with the trace id to find the position from which to purge the table.

Stage RS4 detects and stores traces. RS4 has two buffers dedicated to creating the input and the output contexts and a buffer to store branch bitmaps for a trace being constructed for reuse. Once a trace is constructed, the trace is sent to RS1, where it may be stored in the reuse table. RS4 is also responsible for squashing traces and instructions dependent on a mispredicted trace as they exit the pipeline. This mechanism is the same one used for branch mispredictions.

4. Evaluation

We experimentally evaluated RST to determine its effectiveness and performance benefit when design issues such as the number of wires between stages and chip area are considered. This study investigated how RST improves performance and reuse in comparison to non-speculative trace reuse, how RST's configuration influences performance, whether RST increases demands on processor resources, and how RST compares to alternative designs.

The simulation environment used a modified version of *sim-outorder* from SimpleScalar [1], version 3.0b, modified to model pipelines of varying length. Our simulated processor has 19-stages and an issue width of 4 instructions; the pipeline corresponds to Figure 1. Other architecture parameters for the baseline are shown in Table 1. For most experiments, we simulate two processors: a baseline processor without reuse or prediction, and a processor with speculative trace reuse (RST). The baseline and the RST processor share the same pipeline configuration, except for the inclusion of the RST mechanism. We have also simulated many other baselines with different amounts of L1, L2, and L3 cache (e.g., 32-KB L1 I&D, 512-KB L2, 2-MB L3), pipeline depths (from 6 to 30 stages), and issue widths (up to 8 instructions/cycle). Although the absolute numbers differ, the performance trends for these other pipelines are the same as the results reported in this paper. The RST architecture employed a confidence table with 4096 entries, each one a 2-bit saturated counter.

Table 1. Architecture configuration

Parameter	Value
Pipeline width	4 (2 ALUs, 2 memory, 1 mult)
Pipeline depth	19 (4 fetch, 4 decode, 2 dispatch, 5 issue, 1 execute, 2 writeback, 1 commit)
IFQ, RUU, LSQ	16 instructions, 128 entires, 64 entries
Branch predictor	gshare
First level	13-bit register (xored with PC)
Second level	8192 entries
BTB	4096 entries, 2-way
Instruction set	PISA
L1 cache	32KB I & D, 4-assoc, 64B lines, 1 cycle hit
L2 cache	512KB, 8-assoc, 256B lines, 5 cycles hit
L3 cache	2MB, 8-assoc, 256B lines, 20 cycles hit
Memory	200 cycles first chunk, 20 cycles next chunk

The simulated workload is composed from SPEC CPU 2000 benchmarks, using both the binaries and the reduced input sets from the ARCTiC Labs[10]. These benchmarks and input sets were chosen because they represent both integer and floating-point programs, and can be simulated to completion.

4.1. Performance with reuse table organizations

This first experiment is designed to find a configuration for RST that has good performance and can be easily implemented, addressing the issues with bandwidth from stages RS1 to RS2. Figure 3 shows the speedups over the baseline for different configurations of the reuse tables. The first bar in each set (gray) shows the results for RST with two reuse tables: one for instructions and one for traces. The trace table has 512 entries, with input scopes of 3 registers and output scopes composed of 2 registers. (We also simulated wider scopes, but average performance slightly decreases when more input and output registers are allowed in traces.) The instruction table has 2048 entries. Both tables are 4-way set associative. This configuration provides the best average performance, with a harmonic mean speedup of 1.23 over the baseline architecture without reuse. However, this configuration has significant implementation complexity because the number of wires from stage RS1 to stage RS2 is very large. This size is a function of the number of traces, instructions, the size of the trace input/output scopes, and table associativity. For the studied configuration with 4-way set associative tables, there can be up to 4 candidate traces and 4 candidate instructions for each fetched instruction (i.e., a total of 8 candidates). A candidate trace may have 3 inputs and a candidate instruction may have 2 inputs. For each instruction that is fetched, there are 20 inputs that have to be conveyed between RS1 and RS2. Thus, in a 4-wide issue architecture, 80 inputs, each with 32 bits, has to be transmitted between RS1 to RS2! This number would be even larger if the output contexts were considered as well.

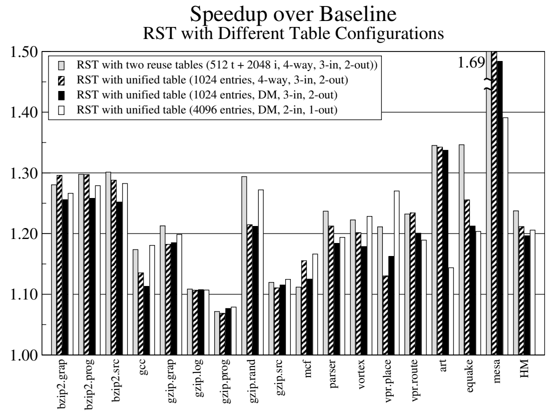


Figure 3. Speedup for RST over baseline with different table configurations

There are two approachs to reduce the bandwidth: (i) decreasing the number of trace/instruction candidates, and (ii) decreasing the size of trace contexts. We explored both approaches. The second bar in each set in Figure 3 shows the results with a single reuse table that can hold traces with at least one instruction (thus being able to reuse instructions too). This configuration conveys fewer candidates between RS1 to RS2: it conveys 4 candidates, rather than 32. In this configuration, we increased the table size to occupy the same area as having the two reuse tables. For most benchmarks, except $bzip2.graph$, mcf, and $vpr.route$, performance decreases slightly when compared to the previous RST configuration with two tables. However, performance is still good, with an average speedup of 1.21 over the baseline. This configuration still demands a large number of values to be transmitted from RS1 to RS2. In this case, there are 4 candidate traces per fetched instruction, each with 3 inputs, leading to a total of 48 values (4-way set associative * 3 inputs * 4-wide issue = 48). The next set of bars (black) depicts results with a direct-mapped reuse table. In this configuration, each PC address would have at most one reuse candidate, hence effectively reducing the number of wires by 75% (only 12 inputs are sent between RS1 and RS2). Again, in most benchmarks there was a decrease in performance as the possibilities of reuse were reduced, but an average speedup of almost 1.21 over the baseline was still achieved.

After reducing the number of tables from two to one, and the associativity from 4 to direct-mapped, the next step was to reduce the size of trace contexts. Each register in the input or output contexts adds 37 bits to a reuse table entry (32 for the value plus 5 for the register index). Reducing from 3 inputs to only 2 inputs and from 2 outputs to only 1 output allows constructing a table with two times more entries, while occupying roughly the same area. The last bar in each set (white) of Figure 3 shows the results for this configuration. Performance was just slightly better than the previous configuration (black bar); however, there are fewer registers in each input context. This further reduces the number of wires from RS1 to RS2. Importantly, in this case, a trace entry with 2 inputs and 1 output occupies as much space in the reuse table as a single instruction in an instruction reuse scheme, without limiting the number of instructions that can be skipped by a single resue table entry. It also has the *same* bandwidth requirements between RS1 and RS2 as a conventional instruction reuse scheme. However, our scheme can reuse and collapse more instructions than instruction reuse in a single cycle. This new configuration with a single, unified reuse table can also speculatively reuse individual instructions in addition to traces.

Thus, we infer from the results presented in Figure 3 that it is possible to significantly reduce the required bandwidth between the reuse stages without losing much performance. In the following sections, we evaluate the results obtained with the final configuration (RST with unified, direct-mapped reuse table with 4096 entries, contexts with 2 inputs and 1 output) in more detail.

4.2. Results with unified reuse table

In the next graph, Figure 4, we show the speedups over the baseline for the direct-mapped, 4096-entry reuse table configuration of RST. All benchmarks had performance improvements over the baseline, ranging from a speedup of 1.08 for $gzip.program$ to 1.62 for $mesa$, and the average speedup is 1.21 (harmonic mean). The results for $mesa$ are very interesting if we consider that it is a floating-point benchmark (whose instructions are not included in RST's reuse domain).

An important question is the amount of useful work that

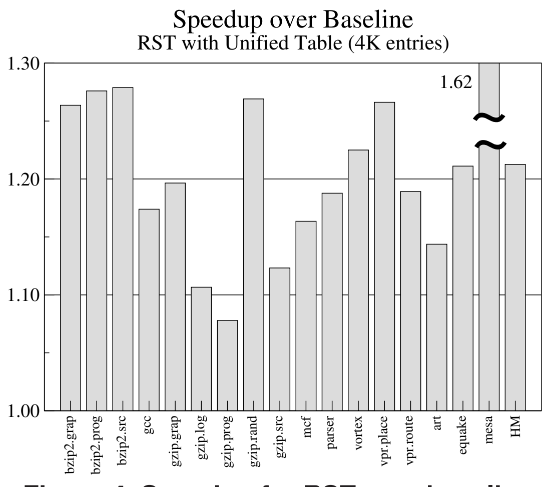


Figure 4. Speedup for RST over baseline

is skipped. Figure 5 shows the contribution of reused instructions to committed instructions. There is a strong correlation between speedup and reused instructions as we expected. In $mesa$, 73% of all committed instructions are provided by the RST mechanism, which explains its large speedup of 1.62. Results for $vpr.place$ and $vpr.route$ are also remarkable. In this case, most of the reuse is actually due to speculative reuse (10.8% and 9.6% of committed instructions). Although the percentage of committed instructions that were speculatively reused is not larger than the other benchmarks, the contribution is comparatively larger when we consider the number of non-speculatively reused instructions. This result indicates that vpr would not get much benefit from non-speculative reuse, because most of the time the inputs for reusable traces were not ready for the reuse test. However, with RST it was able to achieve speedups of up to 1.26 ($vpr.place$).

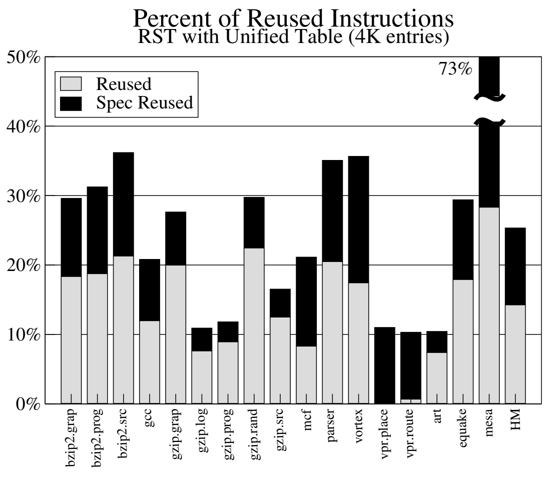


Figure 5. Contribution to reused instructions

After each misspeculated trace, the fetch unit is redirected to the first instruction in the mispredicted trace, like for branch mispredictions. Figure 6 shows the misprediction rates. On average, 9.4% of all speculatively reused traces are mispredicted. The benchmark with more misspeculations is gcc, with 18% of traces being mispredicted. The benchmark $mesa$ has only 6% of mispredicted traces. This rate and the large amount of reused traces (Figure 5) explains why $mesa$ achieve such good performance with RST. The benchmark $gzip.random$ has the smallest misprediction rate, with only 2.3% of mispredictions.

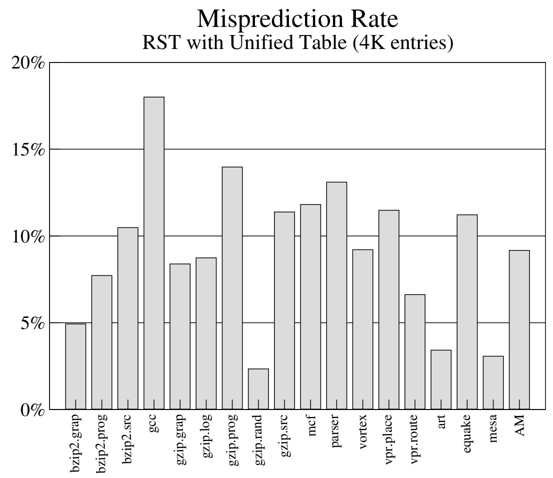


Figure 6. Misprediction rate for RST

Figure 7 depicts the average number of instructions in reused traces. For all benchmarks, except art, speculative reuse has longer traces than non-speculative trace reuse (average of 1.8 instructions per trace against 1.2 for non-speculative reuse). This result demonstrates that as traces grow in length, there is a greater probability that not all inputs will be available for the reuse test. In non-speculative trace reuse, this imposes a severe restriction on average trace length. Adding speculation to the trace reuse mechanism, as done by RST, allows the traces to grow longer and skip more instructions, leading to a performance improvement.

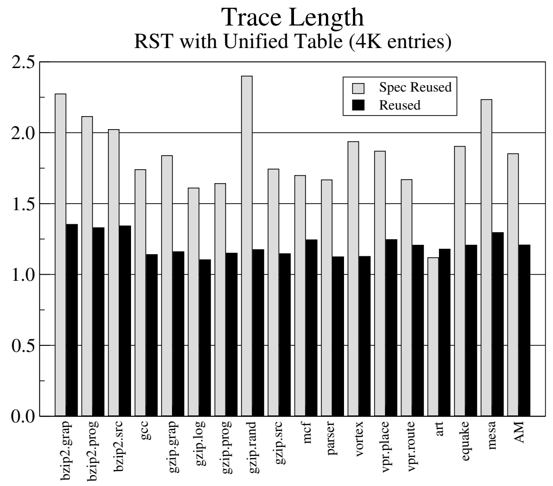


Figure 7. Length of reused traces

4.3. Implementation issues

After reducing the bandwidth from stage RS1 to RS2, we are left with three issues to deal with: (i) access time to reuse table; (ii) storing and testing predictions; and (iii) possible increased pressure in the register file.

First, we measured the access time for the reuse table with CACTI [19], version 3.2. For a direct-mapped 64 KB table and a technology of 0.05 μm, we obtained an access time of 0.61 ns, which could be done in two cycles for a 3 GHz processor. As RST runs a parallel pipeline (Figure 1), we conclude that this would not be an impediment to RST's implementation. Reducing the size of the reuse table can be done, trading size for less performance.

The second issue is how to deal with unresolved predictions. In RST, predictions are stored in the Recovery Table (RT). We measured the number of entries that are required to hold all the predictions at any moment to ensure that RT's size is reasonable. On average, RST required 24% fewer entries than the maximum used for pure value prediction, and for all benchmarks, RST had fewer inflight predictions. This is due to the mixed nature of RST, which is able to both reuse and speculatively reuse values. The average number of inflight predictions is less than 4 predictions, hence a small RT is enough and would not pose a significant problem to be implemented.

Another relevant issue is the pressure on the register file. We measured the number of read and write ports used at each cycle by both the baseline and the RST architectures, and the results show that even though the RST architecture accesses more registers on average, the maximum number of read ports required for RST was larger in only two benchmarks ($gzip.source$ and art) and by only one port (for 99% of time). 7 register read ports are enough for all benchmarks for both architectures. The maximum number of write ports did not noticeably change.

From these experiments, we can infer that the studied issues do not restrict implementation of RST in deeply pipelined superscalar architectures.

4.4. Comparison to alternative schemes

Our final experiments compared RST to alternative schemes, including trace reuse, instruction reuse, and value prediction. For the value prediction experiment, we apply value prediction and reuse to single instructions, including memory operations.

Figure 8 shows the speedups for DTM (Dynamic Trace Reuse, a trace reuse technique [3]) with two tables, DTM with unified reuse table, and RST with unified reuse table over the baseline architecture (without reuse). DTM has good performance when two reuse tables are used, with an average speedup of 1.16 over the baseline. However, with an unified table, reduced context sizes, and a direct-mapped table is used, the speedup decreases to only 1.06. As DTM can only reuse traces with ready inputs, it requires a less speculative trace construction technique and does not adapt well to a unified table scheme. RST has better performance than any of the DTM configurations, with a speedup of 1.21 over the baseline. RST has a small extra cost over DTM with unified reuse table (i.e., the RT and misspeculation detection hardware) but much simpler hardware than DTM with two reuse tables (a fraction of the number of wires and comparators in stages RS1 and RS2).

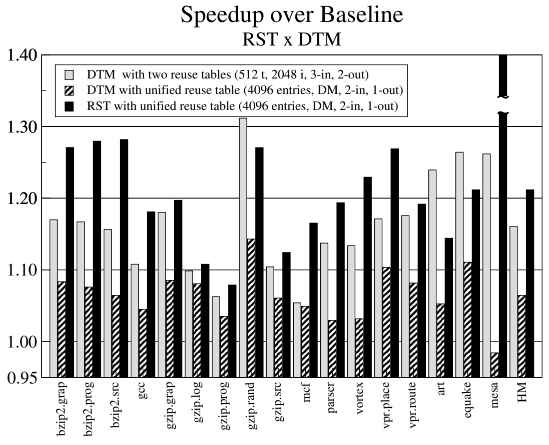


Figure 8. Comparison to DTM

Figure 9 shows the speedups over the baseline architecture for instruction reuse, a value prediction scheme, and RST, using similar table areas. On average, RST outperformed the other mechanisms, with a speedup of 1.21 over the baseline, while value prediction achieved a speedup of 1.20 and instruction reuse only 1.12. For the value prediction simulations, we permitted load instructions to be predicted using an oracle scheme. Thus, the results obtained with this value prediction scheme are expected to be much better than those of an actual implementation. Even with perfect load prediction, our realizable RST implementation still did slightly better than the ideal value prediction scheme.

We also experimented with cache sizes, doubling the size of first level caches from 32 to 64 KB to compare with the same area required to implement RST's reuse table. The results showed only a small increase in performance of 1% when compared to the baseline architecture.

5. Related Work

Many different mechanisms based on value reuse have been proposed. The reuse granularity includes instructions [15, 20], expressions and invariants [13], basic blocks [9], traces [3, 5], as well as instruction blocks and sub-blocks of arbitrary size [8]. The techniques vary in

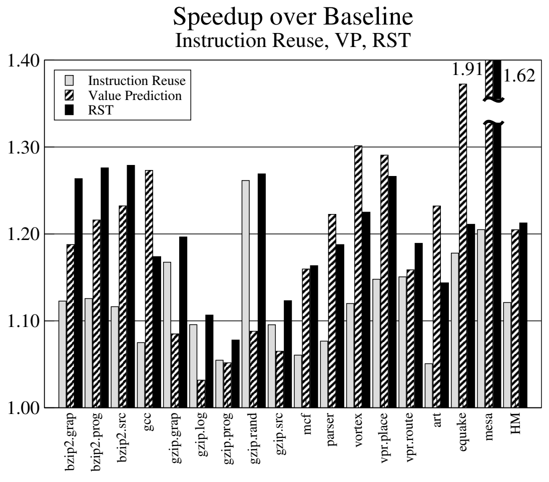


Figure 9. Comparison to Instruction Reuse and Value Prediction

terms of their dependence on hardware and compiler support [8, 22].

Loads and stores are typically not 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 [24]. Another approach uses instruction reuse to exploit both same instruction and different instruction redundancy [23].

Many variations on value prediction have been proposed, including two-level value prediction [21], hybrid value prediction [17, 21], and others, many of which were inspired by branch prediction. Value prediction based on correlation [18, 21] uses global information about the path in selecting predictions. Prediction of multiple values for traces [17] may be done for only the live-out values in a trace, reducing necessary bandwidth in the predictor. Speculation control [6] is used to balance the benefits of speculation against other possibilities. Confidence mechanisms [2] are employed to improve value prediction by restricting prediction of unpredictable values. Past work that explored the limits of value prediction with trace reuse has shown there is much potential for performance improvement [14].

The most similar work related to ours was proposed by Wu *et al* [22]. Comparison between the two works is difficult because that their study uses a compiler-driven simulation. Thus, some constraints may not be reflected in their results. Our approach has the advantage of being independent of special compiler support, ISA changes, and extra execution engine for misprediction recovery.

Huang *et al* proposed a scheme that uses a Speculative Reuse Cache (SRC) and Combined Dynamic Prediction (CDP) to exploit value reuse and prediction [7]. During execution, a chooser picks a prediction from the value predictor or a speculatively reusable value from SRC. With this approach, they have achieved a speedup of 10% in a 16-wide, 6-stage superscalar architecture, with 128 KB of storage for the CDP. Our approach has higher overall speedups and requires less storage.

Liao *et al* [11] combined instruction reuse and value prediction, using a reuse buffer and a value prediction table, that are accessed in parallel. If inputs are unavailable, the value predictor is used. For a 6-stage pipeline, they achieved an average speedup of 9%. Unlike RST, this approach speculatively reuses instructions, rather than traces, and it requires two tables.

6 Summary

In this paper, we presented a study on how a RST implementation can be constrained and still achieves better performance than alternative designs. RST has several advantages over other value reuse and speculation techniques, including*(i)* minimal extra complexity when compared to non-speculative instruction and trace reuse; *(ii)* no changes to the ISA, which provides support for legacy codes; *(iii)* no need for an extra execution engine to recover from misprediction; *(iv)* does not increase the pressure on resources unlike other value prediction techniques; *(v)* can be combined with other instruction-level parallelism and speculation approaches; and *(vi)* traces may encapsulate critical paths and their reuse can collapse data dependencies in a single cycle.

With an unified reuse table, small input and output contexts, and direct-mapped tables, the number of bits that are transported through pipeline stages is more than 8 times smaller than in previous studies. Yet, we showed that RST with a simple, low cost configuration improved performance by a harmonic mean of 21% for several benchmarks in a 19-stage superscalar for the simulated SPEC2000 benchmarks. RST also outperformed alternative schemes for instruction reuse, trace reuse, and value prediction. RST had good performance even in FP benchmarks, although floating-point instructions are not included in the reuse domain.

We showed that the overhead of RST in terms of both memory and hardware support is practical. The low overhead and performance improvement demonstrated that integrating trace speculation with value prediction is a worthwhile approach to improving trace reuse, and ultimately, performance as well. In a future work, we intend to add FP instructions to the reuse domain, and explore RST as an alternative similar to strands [16] to save power.

References

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

[2] B. Calder *et al.* Selective value prediction. In *26th ISCA*, pp.64–74, 1999.

[3] A. T. da Costa *et al.* The dynamic trace memoization reuse technique. In *9th PACT*, pp.92–99, 2000.

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

[5] A. González *et al.* Trace-level reuse. In *28th ICPP*, pp.30–37, 1999.

[6] D. Grunwald *et al.* Confidence estimation for speculation control. In *25th ISCA*, pp.122–131, 1998.

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

[8] J. Huang and D. J. Lilja. Exploring sub-block value reuse for superscalar processors. In *9th PACT*, pp.100–110, 2000.

[9] J. Huang and D. J. Lilja. Extending value reuse to basic blocks with compiler support. *IEEE Trans. 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 *Workshop for Workload Characterization – ICCD*, pp.83–100, 2000.

[11] C.-H. Liao and J.-J. Shieh. Exploiting speculative value reuse using value prediction. In *Proc. of the 7th Asia-Pacific Conf. on Computer Systems Architecture*, pp.101–108, 2002.

[12] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *29th MICRO*, pp.226–237, 1996.

[13] C. Molina *et al.* Dynamic removal of redundant computations. In *13th ICS*, pp.474–481, 1999.

[14] M. L. Pilla *et al.* The limits of speculative trace reuse on deeply pipelined processors. In *15th SBAC-PAD*, pp.36–44, 2003.

[15] A. Roth and G. S. Sohi. Register integration: A simple and efficient implementation of squash re-use. In *33rd MICRO*, pp.223–234, 2000.

[16] P. G. Sassone *et al.* Static strands: safely collapsing dependence chains for increasing embedded power efficiency. In *2005 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems*, pp.127–136, 2005.

[17] R. Sathe *et al.* Techniques for performing highly accurate data value prediction. *Microprocessors and Microsystems*, 22(6):303–313, Nov. 1998.

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

[19] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Tech Report WRL RR 2001/2, Western Labs, Palo Alto, Aug. 2001.

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

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

[22] Y. Wu *et al.* Better exploration of region-level value locality with integrated computation reuse and value prediction. In *28th ISCA*, pp.98–108, 2001.

[23] J. Yang and R. Gupta. Load redundancy removal through instruction reuse. In *29th ICPP*, pp.61–68, 2000.

[24] S. Önder and R. Gupta. Load and store reuse using register file contents. In *15th ICS*, pp.289–302, 2001.