Limits for a Feasible Speculative Trace Reuse Implementation

Maurício L. Pilla*
PPGINF, Catholic University of Pelotas, Brazil
E-mail: pilla@ucpel.tche.br
*Corresponding author
Bruce R. Childers
Department of Computer Science, University of Pittsburgh, USA
E-mail: childers@cs.pitt.edu
Felipe M. G. França
COPPE, Federal University of Rio de Janeiro, Brazil
E-mail: felipe@cos.ufrj.br
Amarildo T. da Costa
Military Institute of Engineering, Rio de Janeiro, Brazil
E-mail: amarildo@cos.ufrj.br
Philippe O. A. Navaux
Computer Science Institute, Federal University of Rio Grande do Sul,
Porto Alegre, Brazil
E-mail: navaux@inf.ufrgs.br

Abstract: Trace reuse is a powerful technique to dynamically collapse instructions. Traces, i.e., dynamic sequences of instructions, are detected during runtime, and their inputs and outputs are stored in a table. The next time the same address is reached and the inputs are the same, this sequence of instructions can be safely bypassed, and the same outputs are written in registers and memory. One of the major issues with trace reuse is that all inputs must be ready for the reuse test, or the trace cannot be reused. Reuse through Speculation on Traces (RST) adds value prediction to trace reuse, so that traces with inputs that are not ready for early validation can be speculatively reused and validated later in the pipeline. Another important problem is the number of wires that are used to transmit inputs from the reuse table to the reuse test stage, which increases with table associativity and pipeline width. In this paper, we compare the limits of RST with two reuse implementation strategies: one with two reuse tables and high associativity, and another with a direct-mapped, unified reuse table. The unified table organization, considerably simpler to implement, presented a speedup of 1.24 over the baseline with a reduction of less than 4% in performance when compared to the two table approach.

Keywords: value reuse; value prediction; processor architectures.

Biographical notes: Maurício L. Pilla is an Associate Professor in the Computer Science School at the Catholic University of Pelotas, Brazil. His research interests include computer architecture, high performance computing and pervasive computing. Pilla received a BS degree (Computer Science, 1999) and a PhD degree (Computer Science, 2004), both from the Federal University of Rio Grande do Sul (UFRGS), Brazil.

Bruce R. Childers is an Associate Professor in the Department of Computer Science at the University of Pittsburgh. His research interests include compilers and software development tools, computer architecture, power-aware computing, and embedded systems. Childers received a BS degree (Computer Science, 1991) from the College of William and Mary, and a PhD degree (Computer Science, 2000) from the University of Virginia.

Felipe M. G. França is an Associate Professor at the Systems Engineering and Computer Science Program, COPPE/UFRJ. His research and teaching interests are asynchronous circuits, computational intelligence, computer architecture, cryptography and all aspects of parallel and distributed computing. He received his BEE degree (Electronics Engineering, 1981) from the Federal University of Rio de Janeiro (UFRJ), a MSc degree (Computer Science, 1987) from COPPE/UFRJ, and a PhD degree (Computer Science, 2002) from COPPE/UFRJ.
1 INTRODUCTION

Control and data dependencies are major impediments to obtaining better performance from modern processor architectures. Simply increasing available resources to improve performance may not be the best choice because an increase in hardware complexity can degrade clock rates. Data dependencies also limit instruction-level parallelism, mitigating performance gains from using better branch prediction, larger caches, deeper pipelines, and more functional units. Therefore, new techniques that can successfully overcome these limits and are not overly complex to implement are necessary to further improve processor performance.

Programs have a large amount of redundant or predictable computation (Bodik et al., 1999; Lipasti and Shen, 1996; Gabbay and Mendelson, 1996; Sazeides and Smith, 1997; Sodani and Sohi, 1998). Based on these observations, value reuse and value prediction can enhance performance and better exploit instruction-level parallelism. Value reuse allows a computation (e.g., an instruction or an instruction 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 a large number of 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. It is typically done to speculate beyond dependencies for individual high-latency instructions, such as memory loads. But value prediction can significantly increase pressure on machine resources, limiting its benefit.

A main issue regarding value reuse methods is how to deal with the extra hardware requirements implied by fetching reuse candidates from a reuse table and testing them every cycle. Each fetched instruction is searched in the reuse tables, and reuse candidates may have more than one input value to be transmitted from the reuse table to where the reuse tests are done. So, an instruction may have more than one reuse candidate, depending on the reuse table associativity. For each reuse candidate to be tested, there are wires from the reuse table to the reuse test stage, and also a comparator. The original design of Reuse through Speculation on Traces (Pilla et al., 2003) follows its non-speculative predecessor, Dynamic Trace Memoization (??da Costa et al., 2000) by using two reuse tables, one for storing reusable instructions, and another for reusable traces of instructions, which further increases the number of candidates, wires, and comparators.

In a previous work (Pilla et al., 2006), these issues were approached by unifying reuse tables and reducing table associativity. Hence, less candidates are sent to the reuse test stage, reducing complexity. In Pilla et al. (2003), limits for reuse through speculation on traces in an architecture with two reuse tables was studied.

In this paper, we study the impact of constraining reuse tables on the upper bound performance of RST. The main contribution of this paper is the comparison of upper bound performance limits of two different implementations. For that comparison, we simulate a two reuse table organization and multiple reuse candidates for each instruction, and a simplified organization with an unified reuse table and a single candidate for each instruction.

This paper is divided into the following sections. First, Reuse through Speculation on Traces is presented in Section 2. Section 3 experimentally evaluates and compares both organizations. Section 4 discusses related work and Section 5 show the main conclusions from this work.

2 RST

RST (Reuse through Speculation on Traces) 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 (??da Costa et al., 2000; González et al., 1999) 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 (Pilla et al., 2003) 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 (Lipasti and Shen, 1996;
Sazeides and Smith, 1997). 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 (Gabbay and Mendelson, 1996). Because RST uses existing information needed by non-speculative trace reuse, it incurs only a small additional hardware cost.

Traces are constructed by recording instructions in the reuse domain as they are committed in a reuse table called Memo_Table_T. Figure 1 shows an example of trace construction and reuse. As a trace is constructed (Figure 1(a)), its input and output contexts (i.e., the trace live-in and live-out register names and values) are determined 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. Figure 1(b) shows the resulting input and output contexts for that specific execution. The next time instruction i1 is fetched, the reuse table is searched and the already stored trace is fetched for the reuse test. If all registers in the input context present the same value as the current architecture state, then the trace is reused. If some of the registers are not ready, the trace may be speculatively reused. In both cases, the output context updates the registers in the architecture, fetch is redirected to instruction i13, and branch prediction is updated with the branch mask of the trace (branches b3 and 8).

A trace is terminated when (i) an instruction outside the reuse domain is encountered, (ii) the trace needs more resources than available (e.g., a maximum number of input/output contexts has been reached), or (iii) a load or store is encountered. A previous limit study showed that including memory operations in the reuse domain does not increase performance enough to warrant the complexity (Pilla et al., 2003). Multiple traces may be memoized in Memo_Table_T that begin with the same instruction (when the table is set associative). During reuse, however, one trace is selected based on the input context. Figure 2 shows the structure of an entry in Memo_Table_T. The address of the first instruction of a trace is stored in pc, and the address of the next instruction after the trace is stored in npc. The npc field is used to set the PC and to skip the instructions belonging to the trace when it is reused. Fields icv and icr store the input values and register indexes, while ocv and ocr store the output values and register indexes. bm is a bitmap to mark branches that occur inside the trace, while btk bits are set when branches are taken. These two last fields are used to update the branch prediction.

Figure 1: Trace construction and reuse in RST.

2.1 RST’s pipeline

The RST architecture is organized as a pipeline parallel to the execution pipeline, as shown in Figure 3. Each reuse stage is responsible for part of the reuse process. The first stage, RS1, reads and stores reuse candidate in reuse tables (Memo_Table_T for traces and Memo_Table_G for instructions). The RS2 stage tests candidates for reuse, speculative or not. The RS3 stage tests for misspeculations, while the last stage, RS4, identifies and builds new traces based on committed instructions.
Figure 4 shows another possible organization for RST. The main difference between this organization and the one shown in Figure 3 is the simplification of reuse tables. This organization 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 simplified, which has only to identify traces. 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.

The two stages that were effectively changed when the reuse tables were unified are RS1 and RS2. Stage RS1 fetches reuse candidates from the reuse table, using the current program counter to address it, and entries in the reuse table are then sent to RS2. When two reuse tables are used, both instruction and trace candidates may happen, and these are sent to RS2 for the reuse test.

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. However, an unified reuse table decreases the number of candidates by half, regardless of the associativity.

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. Reducing the number of candidates also decreases the need for comparers in this stage, and simplifies the logic required to route register values from the register file.

More details about how RST works can be found in (Pilla et al., 2006).

### 3 EVALUATION

We experimentally evaluate RST with two reuse tables and an unified reuse table. We compare performance between them and also against a baseline architecture without reuse.

The simulation environment uses a modified version of *sim-outorder* from SimpleScalar (Burger and Austin, 1997), 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 4. Other architecture parameters for the baseline are shown in Table 1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline width</td>
<td>4 (2 ALUs, 2 memory, 1 mult)</td>
</tr>
<tr>
<td>Pipeline depth</td>
<td>19 (4 fetch, 4 decode, 2 dispatch, 5 issue, 1 execute, 2 writeback, 1 commit)</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>gshare</td>
</tr>
<tr>
<td>Instruction set</td>
<td>PISA</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32KB I &amp; D, 4-assoc, 64B lines, 1 cycle hit</td>
</tr>
<tr>
<td>L2 cache</td>
<td>512KB, 8-assoc, 256B lines, 5 cycles hit</td>
</tr>
<tr>
<td>L3 cache</td>
<td>2MB, 8-assoc, 256B lines, 20 cycles hit</td>
</tr>
<tr>
<td>Memory</td>
<td>200 cycles first chunk, 20 cycles next chunk</td>
</tr>
</tbody>
</table>

We simulate three processors: a baseline processor without reuse or prediction, a processor with RST that uses two reuse tables and 4-way table associativity, and another with a direct-mapped, unified reuse table. The baseline and the RST processors share the same pipeline configuration, except for the inclusion of the RST mechanism.

The RST processors employed perfect confidence estimation to decide when to pursue speculative reuse to measure the upper bounds for speculative trace reuse for both table organizations. Table 2 shows the characteristics of the reuse tables for the RST processors. The area required for both organizations with these parameters is roughly the same. Note that RST with the unified reuse table has N/A for every parameter in the *insns* table because it does not have a dedicated table for reusable instructions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>RST 2 tables</th>
<th>RST unif. table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traces</td>
<td></td>
<td></td>
</tr>
<tr>
<td>associativity entries</td>
<td>4</td>
<td>4096</td>
</tr>
<tr>
<td>inputs outputs</td>
<td>512</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Traces</td>
<td></td>
<td></td>
</tr>
<tr>
<td>associativity entries</td>
<td>4</td>
<td>N/A</td>
</tr>
<tr>
<td>inputs outputs</td>
<td>2048</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

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

In the following subsections, performance and aspects that contribute for the differences in performance between processors with and without reuse are studied.

### 3.1 Trace length

Trace length of reused traces show how many instructions have been reused each time on average. Reusing more instructions at each reuse test may mean that longer data dependencies are collapsed each time. However, longer traces are less likely to be reused, because they require
more inputs to be redundant. Figure 5 presents reused trace length for both organizations, with two and single reuse tables, but only for traces that were not speculatively reused. On average, reused trace length is reduced from 2.3 to 1.2 instructions per trace.

Figure 6 shows the same measurement for speculatively reused traces, with an average of 3.1 to 1.8 instructions per trace. For both reused and speculatively reused traces, there is a tendency towards shorter traces with the unified reuse table organization. This is due mainly because traces with one instruction are not allowed in the organization with two tables, because there is a dedicated table for isolated instructions, Memo_Table_G. When an unified table is employed, then these isolated instructions are stored in the Memo_Table_T, reducing the average trace length. This reduction in trace length causes some impact on performance, as it will be seen later in this paper, but it is compensated by reusing more traces.

Another interesting trend is that if a good confidence mechanism is employed, there is room for speculatively reusing traces longer than non-speculatively reusing traces. Longer traces tend to have more inputs than shorter traces, and thus they have more values that may not be ready for the reuse test.
3.2 Percent of speculative reuse

Figure 7 presents the percentage of traces that were speculatively reused from the total number of reused traces. The last two bars present the arithmetic mean (AM) for each organization. Speculative trace reuse is reduced for all benchmarks, except art, discussed in subsection 3.3. The unified reuse table works also has an instruction reuse table. Thus, the increase in the percentage of non-speculatively reused traces also includes the migration from instructions that were previously reused from Memo_Table_G. On average, 47% of all trace reuse is speculative in the case of the separated reuse tables organization, and 33% of trace reuse for the unified reuse table.

3.3 Contribution to committed instructions

The next graph (Figure 8) depicts the percent of all committed instructions that were reused as isolated instructions or inside traces. All benchmarks presented a larger percent of instructions from traces when the unified reuse table was employed, but only art presented a larger rate of reused instruction for this organization. This behavior is related to the percent of speculative reuse (see Figure 7), suggesting that many isolated instructions were not reused in the other reuse table organization.

3.4 Performance comparison

In this last measurement, we wanted to verify how much hardware constraints would impair RST’s ability to exploit redundancy and predictability in superscalar processors. Figure 9 shows the speedup over the baseline architecture without reuse for both RST organizations, with two reuse tables and an unified table. The vertical axis present the speedup, while the horizontal axis shows each benchmark and the harmonic mean of speedups (HM). The black bars are related to the 4-way associative, two reuse tables (one reuse table for single instructions and a separated table for traces), while the gray bars show results for RST with a single, direct-mapped reuse table.

For most benchmarks, two reuse tables with high associativity produce better results, although three benchmarks (gcc, mcf and mesa) achieved better performance with the unified table approach. This better performance shows that not only the number of reused traces determines performance, but other aspects such as data dependency chains also play an important role. On average, speedup decreased from 1.28 to 1.24.

Figure 10 shows the IPC ratio between RST with unified reuse table and RST with two reuse tables. On average, RST with an unified reuse table presents 3.9% less performance than the previous organization. However, it still presents a good performance improvement over not reusing traces, as it was shown in Figure 9.

3.5 Remarks

At this section, we discussed experimental results from two reuse table organizations. We found that reducing associations or inside traces. All benchmarks presented a larger percent of instructions from traces when the unified reuse table was employed, but only art presented a larger rate of reused instruction for this organization. This behavior is related to the percent of speculative reuse (see Figure 7), suggesting that many isolated instructions were not reused in the other reuse table organization.
Figure 9: Speedup over baseline for RST with two reuse tables and unified table.

Figure 10: Performance comparison between RST with unified and two reuse tables.

tivity, unifying reuse tables, and restricting the number of inputs and outputs result in less performance and shorter traces. However, this restricted organization still provides good speedups over the baseline architecture without reuse, with an upper bound average speedup of 1.24, and has the advantage of being simpler to implement.

4 Related Work

Many different mechanisms based on value reuse have been proposed. The reuse granularity includes instructions (Roth and Sohi, 2000; Sodani and Sohi, 1998), expressions and invariants (Molina et al., 1999), basic blocks (Huang and Lilja, 2000b), traces (da Costa et al., 2000; González et al., 1999), as well as instruction blocks and sub-blocks of arbitrary size (Huang and Lilja, 2000a). The techniques vary in terms of their dependence on hardware and compiler support (Huang and Lilja, 2000a; Wu et al., 2001).

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 (Önder and Gupta, 2001). Another approach uses instruction reuse to exploit both same instruction and different instruction redundancy (Yang and Gupta, 2000).

Many variations on value prediction have been proposed, including two-level value prediction (Wang and Franklin, 1997), hybrid value prediction (Sathe et al., 1998; Wang and Franklin, 1997), and others, many of which were inspired by branch prediction. Value prediction based on correlation (Sazeides and Smith, 1997; Wang and Franklin, 1997) uses global information about the path in selecting predictions. Prediction of multiple values for traces (Sathe et al., 1998) may be done for only the live-out values in a trace, reducing necessary bandwidth in the predictor. Speculation control (Grunwald et al., 1998) is used to balance the benefits of speculation against other possibilities. Confidence mechanisms (Calder et al., 1999) 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 (Pilla et al., 2003).

The most similar work related to ours was proposed by Wu et al. (2001). 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 and Lilja (1999) proposed a scheme that uses a Speculative Reuse Cache (SRC) and Combined Dynamic Prediction (CDP) to exploit value reuse and prediction. 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 and Shieh (2002) 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.

5 CONCLUSION

In this paper, we presented a comparison of two alternative organizations for reuse tables. Although the more complex organization has better performance, with a speedup of 1.28, its tables are more complex to implement, and more wires from the tables to the reuse test stage are required. Hence, a new approach with an unified reuse table is a more feasible organization, with a speedup of 1.24 and a small cost of 4% in performance when compared to the
complex organization. A desirable side effect of the unified reuse table is that it allows for single instructions to be speculatively reused as well as traces. In future work, we want to investigate how an unified reuse table would fit a non-speculative trace reuse architecture such as Dynamic Trace Memoization (da Costa et al., 2000).

ACKNOWLEDGMENT

This work was partially supported by CNPq, CAPES, and FAPERGS grants.

REFERENCES


