The Dynamic Trace Memoization Reuse Technique

Amarildo T. da Costa, Felipe M. G. França, Eliseu M. C. Filho
COPPE/Federal University of Rio de Janeiro
Department of Systems and Computer Engineering
P.O. Box 68511 21945-970 Rio de Janeiro, Brazil
{amarildo, felipe, eliseu}@cos.ufrj.br

Abstract

Dynamic Trace Memoization (DTM) is a reuse technique that employs memoization tables to skip the execution of sequences of redundant instructions. For the SPECInt95 benchmark programs, DTM delivers performance improvements from 5% to 21% with an average of 9.3%. Moreover, DTM attains twice the average speedup of two other previously proposed reuse mechanisms for a subset of the SPECInt95 benchmarks.

1. Introduction

Redundant instructions represent a significant portion of the instructions executed by a program [1]. Redundant instructions are dynamic instances of the same static instructions which execute with the same operand values and therefore produce the same result. Instruction redundancy is originated, for example, in expressions within loops or procedures that repeatedly compute upon the same or quasi-identical data.

Compiler transformations like common subexpression elimination and loop-invariant code motion [2] may be not effective in finding redundancies that manifest themselves at run-time. *Function Memoization* [3] is an example of a run-time software technique for redundancy elimination. By using *memoization tables*, it allows functions without side effects to immediately return values already known from previous executions with the same input arguments. Our scheme is directly inspired by Function Memoization.

Redundancy elimination in hardware [4, 5, 6] is accomplished by caching the results of redundant instructions within the processor and reusing them on later occurrences of the instructions. This paper presents a hardware-based reuse technique called DTM – *Dynamic Trace Memoization* [7]. DTM is able to reuse sequences (traces) of redundant instructions *and* individual redundant instructions as well.

Let us define the *actual reuse* as the fraction of the available redundancy which is in fact exploited by a given reuse mechanism. For the SPECInt95 benchmarks, DTM achieves an average actual reuse of 44%, compared to 19% for the S_{n+d} reuse mechanism [4, 5] and 22% for the Block Reuse mechanism [6]. For the largest subset of SPECInt95 benchmarks common to DTM, S_{n+d} and Block Reuse evaluation studies, DTM introduces an average speedup of 10.2% whereas S_{n+d} with the same bit storage capacity and Block Reuse with twice the bit storage capacity provide average speedups of 4.4% and 3.4%, respectively.

This paper is organized as follows. Section 2 provides a functional description of the DTM technique. Section 3 shows how DTM could be incorporated into a typical superscalar microarchitecture. Section 4 reviews the related works. Section 5 presents the evaluation results. Section 6 summarizes the main conclusions and identifies future works.

2. Dynamic Trace Memoization

A *trace* is a dynamic sequence of instructions issued during the execution of a program. A trace is redundant if it is only comprised of redundant instructions. On the contrary to other reuse schemes, redundant traces in DTM do not include `LOAD` and `STORE` instructions. However, DTM does reuse address calculations associated to memory access instructions.

The key aspects in the operation of DTM are: (1) how redundant instructions are detected; (2) how redundant instructions are utilized to construct trace instances; (3) how redundant trace instances are identified; and (4) how redundant traces and redundant instructions are reused. These issues are discussed next.

2.1. Detection of Redundant Instructions

Detection of redundant instructions is the initial step in both trace construction and single-instruction reuse. It involves verifying whether the current dynamic instance of a given static instruction uses the same operand values seen by a previous instance. For such purpose, DTM employs a fully-associative table called *Global Memoization Table*, or *Memo_Table_G*.

Each *Memo_Table_G* entry keeps information associated to a dynamic instruction. A *pc* field contains the static instruction's address. Two fields, *sv1* and *sv2*, hold the operand values used by the dynamic instruction. A *res/targ* field holds either the result of an arithmetic-logical instruction or the target address of a control transfer instruction. Two bits, *jmp* and *brc*, identify the instruction as a jump or branch, respectively. Another bit, *btaken*, indicates whether the branch was taken or not. Note that *Memo_Table_G* does not keep instruction codes.

Each dynamic instruction is classified as either redundant or non-redundant, as follows. Invalid instructions (instances of `LOAD` or `STORE`) are simply tagged as non-redundant. The address and current operand values of a valid dynamic instruction are associatively compared against the *pc*, *sv1* and *sv2* fields in the *Memo_Table_G* entries. If a match does not occur, then the instruction is tagged as non-redundant, an entry in *Memo_Table_G* is allocated and the fields *pc*, *sv1* and *sv2* are filled appropriately. If a match occurs, then the instruction is tagged as redundant but a *Memo_Table_G* entry is not allocated. If the dynamic instruction is non-redundant, then the ongoing trace construction is finished. If the instruction is redundant, the input and output contexts of the trace currently under construction are updated, as described next.

2.2. Trace Construction

The *input context* of a trace is defined as the set of operand values used by instructions forming the trace, but which are produced by instructions outside the trace. The *output context* of a trace is the set of results produced by instructions within the trace. The construction of a trace consists of adding values to the trace's input and output contexts for each redundant instruction found.

Figure 1 shows the structures employed to assemble and store traces. The *input context map* and *output context map* have a bit for each architectural register. For each redundant instruction, a bit of the input context map is set if the corresponding source register holds a value which was last written by an instruction outside the trace being constructed. The bit of the output context map corresponding to the instruction's destination register is also set. Whenever a bit of the input (output) context map is set, the respective register contents is added to the input (output) context of the trace under construction.

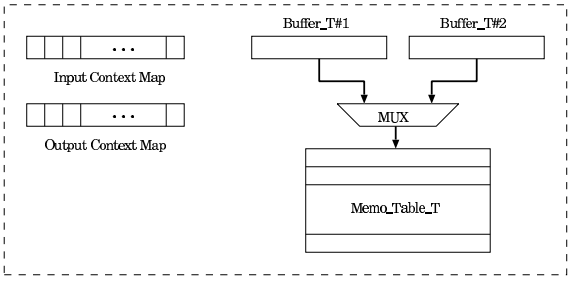

Figure 1. Structures for trace storage.

Buffers *Buffer_T#1* and *Buffer_T#2* hold information about the trace under construction. They allow the construction of a new trace to start before information for the previous trace has been saved. Once the construction of a trace is complete, trace information is transferred to a fully-associative table called *Trace Memoization Table*, or *Memo_Table_T*. Figure 2 depicts the format of *Memo_Table_T* entries (note that the temporary buffers have the same format).

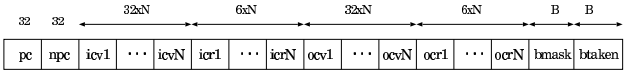

Figure 2. Format of *Memo_Table_T* entries.

The field *pc* stores the address of the initial instruction, while the field *npc* specifies the address of the next instruction to be executed in case the trace is reused (*npc* reflects all control transfers determined by jump/branch instructions within the trace). The *icv* fields hold the trace's input context values, while the *icr* fields identify the corresponding source registers (as indicated by the input context map). The *ocv* fields store the trace's output context values, while the respective destination registers (indicated by the output context map) are specified by the *ocr* fields. Although not shown in Figure 2, each register identifier field has a validity bit. The bits in *bmask* indicate a branch instruction within the trace; the corresponding bit in *btaken* indicates whether the branch was taken or not.

It is important to realize that *Memo_Table_T* does not store instruction codes. Also, note that *Memo_Table_T* may keep multiple instances of the same trace, each instance with a different input context.

In Figure 2, the *context size* N specifies the maximum number of elements in the input and output contexts. The *branch limit* B indicates the maximum number of branch instructions within the traces. The construction of a trace finishes whenever one of the following events occur: (1) a non-redundant instruction is found; (2) the number of context elements has reached the limit N; or (3) the number of branch instructions within the trace has reached the limit B.

2.3. Detection of Redundant Traces

Memo_Table_T is employed to detect redundant trace instances, as follows. The address of each dynamic instruction is associatively compared against the *pc* fields in *Memo_Table_T* entries. For those entries with matching *pc* fields, the current contents of the registers pointed by the (valid) *icr* fields are associatively compared against the values stored in the corresponding *icv* fields. The trace is redundant if the current register values match the input context of a previous instance of the trace stored in *Memo_Table_T*.

In microarchitectures with multiple-instruction fetching, different instruction addresses may simultaneously match *pc* fields in *Memo_Table_T*. This means that instances of different traces are candidates to be redundant (each distinct address represents a different trace). In this case, redundancy check is performed only for instances of the *same* trace. This is in order to keep the implementation complexity of DTM acceptable (see Section 3).

2.4. Reusing Single Instructions or Traces

As explained in Section 2.1, *Memo_Table_G* is employed to determine whether a given dynamic instruction is redundant or not. We say that a *Memo_Table_G* hit occurs whenever a redundant instruction is detected. As seen in Section 2.3, *Memo_Table_T* is utilized to identify redundant trace instances. We say that a *Memo_Table_T* hit occurs if a redundant trace instance is found.

The decision of reusing either a single instruction or a trace is based on the occurrence of *Memo_Table_G* and/or *Memo_Table_T* hits. A single instruction is reused if a hit occurs only in *Memo_Table_G*. In this case, the instruction result is obtained from the *res/targ* field in the *Memo_Table_G* entry corresponding to the redundant instruction. A trace is reused whenever a *Memo_Table_T* hit occurs, even in the case a hit occurs in both memoization tables. The output context values (in *ocv* fields, see Figure 2) are written into the appropriate registers (as indicated by the valid *ocr* fields). In addition, the address in the *npc* field is loaded into the program counter and branch prediction state may be updated with information in the *btaken* field.

3. A Microarchitecture with DTM

This section describes a DTM implementation for an existing superscalar microarchitecture. Except for a few differences, the substrate microarchitecture considered here is similar to that embodied in the AMD K6-III processor [8]. Figure 3 shows the relevant components of the microarchitecture.

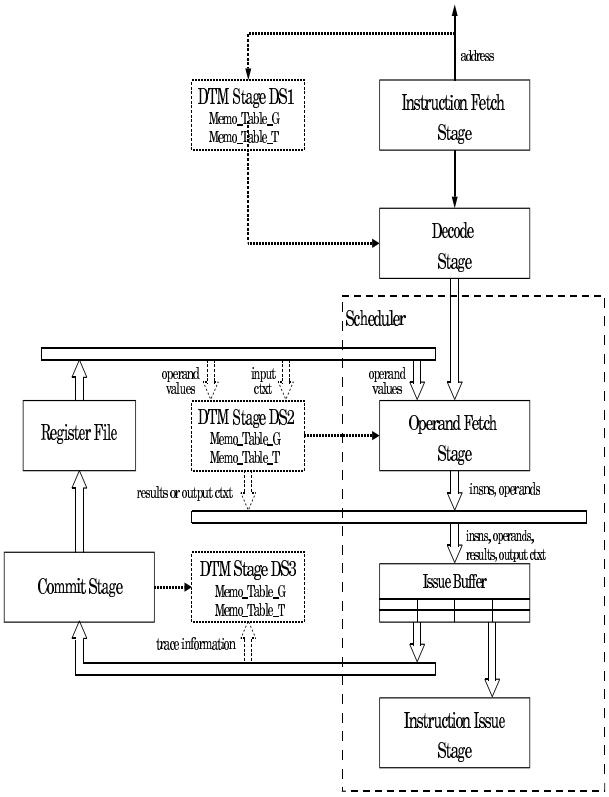

Figure 3. A microarchitecture with DTM.

The Operand Fetch Stage reads valid operand values from the register file and inserts instructions with their operands into the top row of the Issue Buffer. The Issue Buffer provides both centralized reservation stations and reorder buffer functionality. Each Issue Buffer entry is also an implicit renaming register [8], therefore there is no distinction between architectural and physical registers. The Issue Stage selects instructions in the Issue Buffer for execution according to operand availability. Instructions are committed from the bottom row of the Issue Buffer.

The DTM mechanism is organized into three pipeline stages, *DS1*, *DS2* and *DS3* which operate in parallel with the Instruction Fetch, Operand Fetch and Commit stages. Figure 3 indicates the DTM stages and datapaths (in dotted lines) and the memoization tables accessed by each DTM stage. The next three operations are related with trace construction.

(1) Inserting instructions into Memo_Table_G. DS1 allocates a *Memo_Table_G* entry for each valid instruction fetched (LRU replacement is assumed). DS1 provides the indexes of newly allocated entries to the Decode Stage, which attaches each index to the respective instruction before dispatch. DS3 snoops the address, operand values and result of each instruction read by the Commit Stage and fills the *Memo_Table_G* entry pointed by the index attached to the instruction. If the instruction is squashed, then DS3 releases the appropriate *Memo_Table_G* entry. Therefore DTM captures only instructions along *correct* execution paths.

(2) Finding redundant instructions. DS1 compares the address of each fetched instruction against the *pc* fields in *Memo_Table_G* and selects those entries with matching *pc*'s. DS2 snoops the valid operand values read by the Operand Fetch Stage and compares them against the contents of the *sv1* and *sv2* fields in those *Memo_Table_G* entries pre-selected by DS1. If a hit occurs, then DS2 indicates to the Operand Fetch Stage that the corresponding instruction is redundant.

(3) Updating trace context. DS3 snoops the source register id's, the operand values, the destination register id and the result of each instruction committed. If the instruction is redundant, then DS3 updates the context bitmaps and fills the appropriate fields in one of the temporary buffers. If the instruction is not redundant, then DS3 finishes trace construction by saving trace information into a *Memo_Table_T* entry.

The next operations are related with instruction and trace reuse.

(4) Selecting candidate redundant instructions and traces. DS1 compares each fetch address against the *pc* fields in both *Memo_Table_G* and *Memo_Table_T* and selects those entries with matching *pc*'s. DS1 selects only *Memo_Table_T* entries holding instances of the same trace (see Section 2.3).

(5) Identifying a redundant instruction or trace. DS2 looks for redundant instructions in *Memo_Table_G* as described in (2). In addition, DS2 reads the registers indicated by valid *icr* fields in the previously selected *Memo_Table_T* entries and compares the operand values against the *icv* contents.

(6) Reusing an instruction or a trace. If a hit only occurs in *Memo_Table_G*, then the Operand Fetch Stage dispatches the redundant instruction(s) and result(s) (the latter obtained from *Memo_Table_G*) to the Issue Buffer. If a hit in *Memo_Table_T* occurs, then the Operand Fetch Stage dispatches up to N <*ocr*,*ocv*> pairs into the Issue Buffer. In either case, the involved Issue Buffer entries are not considered for scheduling, but the results they contain can be forwarded to other instructions. Committing these entries is carried out in the usual manner.

Implementation Issues. The critical path of the DTM mechanism is comprised by the operations involved in trace reuse, namely: (1) read current input context values from the register file; (2) compare them against previous input context values stored in *Memo_Table_T*; and (3) write output context values into the Issue Buffer. Operation (1) employs dedicated register file ports (see discussion below), thus it can be performed in parallel with operand read by the Operand Fetch Stage. Operation (3) just replaces normal instruction dispatch. Therefore, DTM operations (1) and (3) are hidden by other operations already performed by the substrate microarchitecture.

However, operation (2) adds a delay to the clock cycle time. The total delay can be partially hidden by instruction cache access time, and the delay still visible may be acceptable depending on the targeted clock cycle. However, for very short clock cycles (clock frequencies close or above 1 GHz), an extra pipeline stage is required for *Memo_Table_T* accesses. The present work evaluates the performance of DTM for the case in which additional pipeline stages are not necessary.

Another issue is the number of register file read ports required by the DTM mechanism. It is important to realize that stage DS2 does not need extra read ports to access the current operands of individual instructions. As explained, DS2 just *snoops* the operands normally read by the Operand Fetch Stage (see Figure 3). However, DTM requires additional read ports in order to allow current input context values to be read concurrently with normal operand fetch.

From Section 2.3, recall that only instances of the *same* trace are selected for redundancy check. Consequently, the number of extra ports is equal to the input context size. As it will be shown in Section 5.2, input contexts have at most four elements for the majority of the redundant traces. Consider the register file in the AMD K6-III, which has nine read ports: extending the K6-III microarchitecture with DTM would then require a register file with thirteen read ports. Such a register file is affordable, both in terms of access time and silicon area, even in current integration technologies [9].

4. Related Works

In [4], Sodani and Sohi propose three reuse schemes named S_v, S_n and S_{n+d}. These schemes rely on a structure called *Reuse Buffer* (*RB*) to save instruction operands and results. Schemes S_v and S_n reuse single instructions, however S_{n+d} is able to reuse chains of interdependent instructions. In [6], Huang and Lilja introduce the notion of *basic block value locality* and present the *Block Reuse* technique. Their idea is to explore broader granularity of reuse, from the level of single instructions to basic blocks. Block Reuse employs a structure named *Block History Buffer* (*BHB*) to store the input and output data sets of basic blocks.

Like S_{n+d} and Block Reuse, DTM primarily seeks to reuse sequences of instructions. However, DTM differs from S_{n+d} and Block Reuse with respect to important aspects. The main differences to S_{n+d} are:

• In S_{n+d}, dependence relationships are evaluated only for those instructions within the same dispatch cycle [4]. As a consequence, the number of instructions that are simultaneously reused is constrained by the dispatch width. In DTM, the number of instructions within a redundant trace is independent of the dispatch width, making it attractive even for microarchitectures with narrow dispatch widths.

• In S_{n+d}, an instruction chain may not be entirely reused in a single cycle depending on the dispatch slot occupied by its first instruction. Given a dispatch width w and an instruction chain with length $l \leq w$, all of the instructions forming the chain will be reused in a single cycle only if the first instruction occupies the dispatch slot $s \leq w - l$. On the contrary, DTM does not suffer from instruction misalignment problems;

• S_{n+d} keeps *RB* entries for all the instructions forming a chain [4]. As a result, the average length of the reused chains may be affected by the arrival of new instructions to the *RB* and by the instruction replacement policy. DTM does not retain the instructions covered by traces. Moreover, redundant instruction gathering does not interfere with trace reuse, as these two processes manipulate distinct memo tables;

The differences with respect to Block Reuse are:

• Block Reuse exploits redundancy within basic block boundaries, whereas DTM is not constrained by code-level boundaries. Crossing the boundaries of basic blocks leads to better reuse potential;

• Block Reuse requires compiler assistance in order to detect and mark dead definitions from basic blocks [6], information which then needs to be conveyed to the hardware. By not requiring compiler support, DTM does not suffer from compatibility problems with legacy code;

• In Block Reuse, the *BHB* stores the input and output values for a basic block from its previous four executions [6]. This requires a large storage capacity and is not cost-effective for basic blocks with short locality histories. DTM manages value locality dynamically, by allocating *Memo_Table_T* entries as necessary to keep new trace instances and by releasing entries in a LRU basis. This results in a better utilization of hardware resources.

In [10], Gonzalez *et al.* propose a trace reuse mechanism which employs a *Reuse Trace Memory* (*RTM*) to keep trace information. However, the issues of how the *RTM* is incorporated into a real microarchitecture are not considered. The work provides the speedup upper bounds achieved with an infinite *RTM*, but it does not quantify performance gains for realistic *RTM* sizes. Our work approaches trace-level reuse by providing an evaluation of a feasible reuse mechanism.

A key difference of DTM with respect to the other reuse schemes mentioned here is the exclusion of memory access instructions from the validity domain. We have preferred not to consider these instructions to avoid incurring into timing and cost penalties related to keeping the memo tables consistent with respect to the data memory.

5. Experimental Evaluation

The out-of-order timing simulator of the SimpleScalar Tool Set 2.0 (MIPS-IV ISA) was modified to include the three DTM stages shown in Figure 3. In order to allow a fair comparison, the architectural parameter values adopted are similar to those assumed in [4] and [6]. The main architectural parameter values are: fetch width of 4 instructions per cycle; dispatch width of four instructions per cycle; reorder buffer with 16 entries; 4 integer funcional units and 2 load/store units.

The SPEC95 integer programs were adopted as the benchmark programs. They were compiled by using the C compiler provided in the SimpleScalar Tool Set (`gcc-2.6.3`) with the -O3 optimization level. All benchmark programs executed to completion for their respective reference input data sets.

Performance gains are measured as the speedup given by the ratio ipc_{DTM}/ipc_{BASE}, where ipc_{DTM} is the *instructions per cycle* or *ipc* rate of the microarchitecture with DTM and ipc_{BASE} is the *ipc* rate of the original microarchitecture. The actual reuse is given by the ratio N_r/N_t, where N_r is the number of instructions reused (either individually or as part of a trace) and N_t is the total dynamic instruction count.

5.1. Performance Gains and Actual Reuse

Figure 4 shows the percentage speedup provided by DTM when *Memo_Table_G* and *Memo_Table_T* have the same size, varying from 128 to 4096 entries.

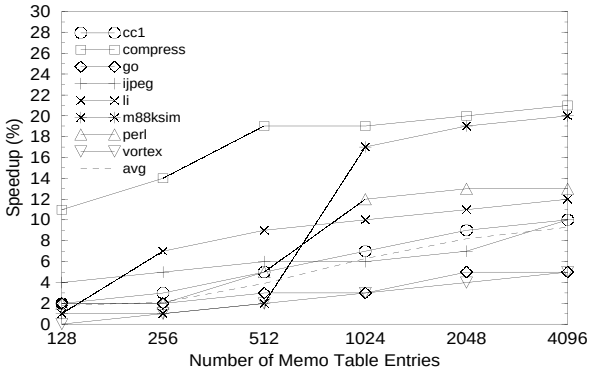

Figure 4. Speedup provided by DTM.

Performance improvements are small for memo tables with only 128 entries, although a speedup of 11% is observed for `compress`. However, substantial gains are observed as the memo table sizes increase to 1024 entries. For memo tables with 1024 entries, the speedup is 10% for `li`, 12% for `perl`, 17% for `m88ksim` and 19% for `compress`; the speedup varies from 3% to 7% for the remaining benchmark programs. Improvements are still noticeable for memo table sizes larger than 1024 entries: for memo tables with 4096 entries, speedups range from 5% (`go` and `vortex`) to 21% (`compress`). The average (harmonic mean) speedup across the benchmark programs is 1.8% for 128-entry memo tables, 6.3% for 1024-entry memo tables and 9.3% for memo tables with 4096 entries.

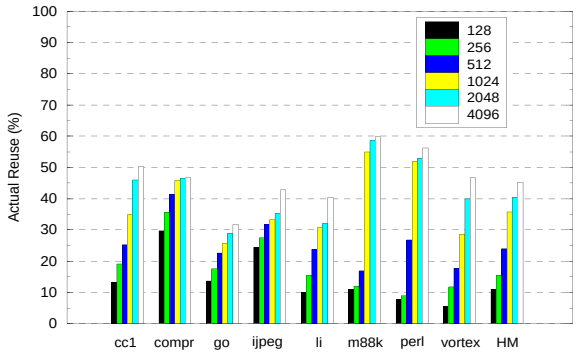

Figure 5. Actual reuse of DTM.

Figure 5 shows the actual reuse given by DTM. With 128-entry memo tables, the actual reuse varies from 5% (`vortex`) to 30% (`compress`). As the memo table size increases to 1024 entries, the actual reuse varies from 26% (`go`) to 55% (`m88ksim`). An actual reuse ranging from 32% (`go`) to 60% (`m88ksim`) is observed for memo tables with 4096 entries.

5.2. Input Context Size and Trace Size

Figure 6 shows the distribution of trace sizes as a percentage of the number of traces actually reused. Most of the reused traces have from three to five instructions. On the average, only 3% of the traces have length equal or greater than six instructions. The benchmarks `go` and `vortex` exhibit the highest occurrence of traces with minimal size (two instructions).

Figure 7 (next page) shows the distribution of input context sizes as a percentage of the number of traces actually reused. Observe that the majority of the traces have input contexts with one to three operands. On the average, only 2% of the traces have input context sizes of four operands. The input context sizes are less than or equal to four elements for 99.6% of the redundant traces (this explains why *Memo_Table_T* configurations with $N = 4$ were assumed). Note that `go` and `vortex`, for which DTM performed worst, are the benchmarks with the highest frequency of zero-operand input contexts (see the discussion on speedup comparison).

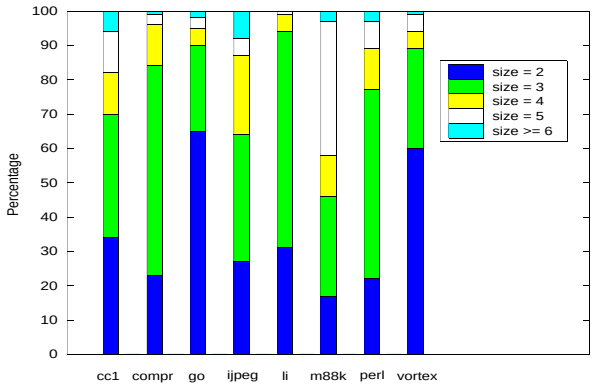

Figure 6. Trace size.

5.3. Comparative Analysis

The speedup and actual reuse achieved by DTM are now compared against the performance gains and actual reuse provided by S_{n+d} and Block Reuse. In order to make a fair comparison, DTM and S_{n+d} will have the same bit storage capacity. In the augmented S_{n+d} described in [5], each *RB* entry has 196 bits (assuming 32-bit tags, 12-bit table indexes, 5-bit register identifiers and 32-bit data), therefore a 4096-entry *RB* has

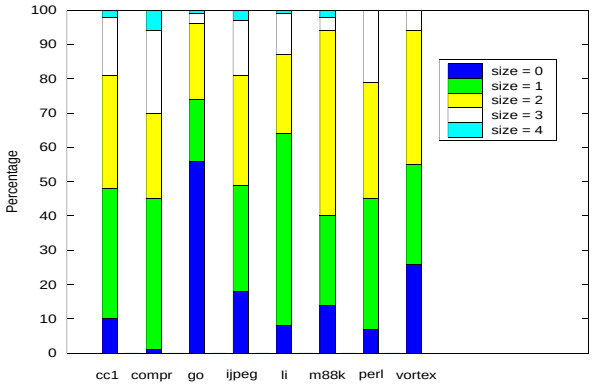

Figure 7. Input context size.

784 Kbits. Each *Memo_Table_G* entry has 131 bits and, assuming $N = B = 4$, each *Memo_Table_T* entry has 376 bits; altogether, both memo table entries require 507 bits. Therefore, DTM with 1583-entry memo tables also requires 784 Kbits of storage capacity.

For the Block Reuse mechanism, a 4/4/3/2 2048-entry *BHB* has the *smallest* bit storage capacity among those for which experimental data provided in [6]. In such a configuration, each *BHB* entry has 845 bits (assuming 32-bit tags, 5-bit register identifiers and 32-bit data), hence a 2048-entry *BHB* has a total of 1,690 Kbits or 2.16 times the bit storage capacity of the DTM and S_{n+d} configurations.

Figure 8 shows the percentage speedups provided by DTM, S_{n+d} and block reuse. Speedups for S_{n+d} and Block Reuse shown here are reproduced from [5] and [6], respectively (speedups are not reported for `li` in [5] nor for `cc1` and `vortex` in [6]).

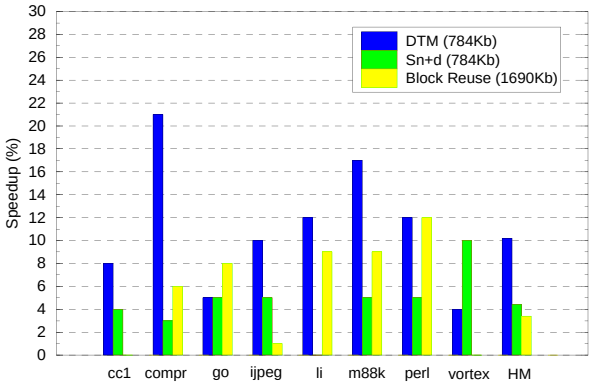

Figure 8. Speedup comparison.

For most benchmark programs, the performance improvements attained by DTM are significantly greater than those achieved with S_{n+d} and block reuse. The only exceptions are: (1) S_{n+d} attains the same performance as DTM for `go` and outperforms DTM for `vortex`; and (2) block reuse has the same performance as DTM for `perl` but is better than DTM (and S_{n+d}) for `go`. The DTM average (harmonic mean) speedup over *all* the benchmark programs is 8.4%. Considering only the subset of benchmark programs for which evaluation data is reported in both [5] and [6], DTM achieves an average speedup of 10.2% while S_{n+d} and block reuse provide average speedups of 4.4% and 3.4%, respectively.

In the case of `go`, the lower DTM performance comes mainly from the fact that 55% of the reused traces have input contexts with size zero (see Figure 7), meaning that the traces are mostly comprised by instructions that load immediate values into their destination registers. As the operand values do not come from registers, there are very few true data dependences among the instructions. Therefore, the ability of trace reuse to collapse data dependence chains has a smaller weight in this case and, consequently, the impact on performance is also smaller.

In the case of `vortex`, the lower DTM speedup is explained mainly by fact that memory accesses represent 53% of the instructions executed by this benchmark program. As DTM does not include memory access instructions in its validity domain, the proportion of redundant instructions reused is smaller than in the other programs.

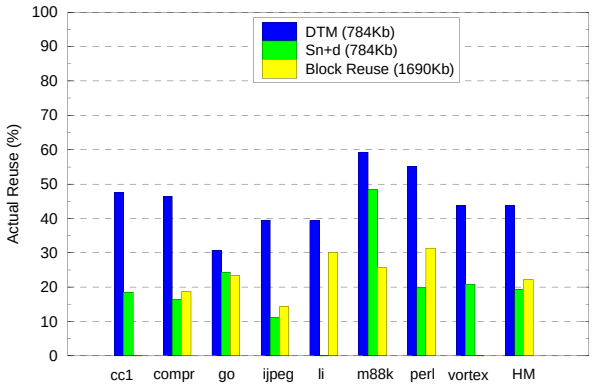

Figure 9. Actual reuse comparison.

Figure 9 shows a comparison of the actual reuse attained by DTM, S_{n+d} and Block Reuse. The actual reuse values for S_{n+d} and Block Reuse are those reported in [5] and [6], respectively. DTM is able to reuse considerably more instructions than S_{n+d} and Block Reuse. Actual reuse provided by DTM varies from 31% (`go`) to 59% (`m88ksim`), whereas the actual reuse ranges from 11% to 48% for S_{n+d} and from 14% to 31% for Block Reuse. On the average (harmonic mean across the benchmark programs common to the three

studies), DTM reuses 44% of the dynamic instructions, compared to 19% and 22% by S_{n+d} and Block Reuse, respectively. According to [10], the average (geometric mean) number of redundant instructions executed by the SPEC95 integer benchmarks accounts for 83% of the dynamic instructions.

From Figure 8, recall that DTM and S_{n+d} performed equally for the go program and that S_{n+d} outperformed DTM for the `vortex` program. But, in Figure 9, note that DTM reuses more instructions than S_{n+d}, even in the case of these two programs. This apparent inconsistency comes from the fact that the volume of reused instructions in DTM is not the only factor determining speedup, trace characteristics are equally important. For example, input context size – which is in connection with critical path length – and trace size – which reflects the number of instructions simultaneously bypassed – should also be taken into account when explaining end performance.

6. Conclusions and Future Works

This work introduced Dynamic Trace Memoization (DTM) as a technique to reuse sequences of redundant instructions. For most SPECInt95 benchmarks, DTM performed better than S_{n+d} when both schemes have the same bit storage capacity. DTM outperformed Block Reuse even when the latter has twice the bit storage capacity.

Several aspects of the DTM technique are now being investigated. As mentioned, the three-stage mechanism depicted in Figure 3 is not adequate for very short clock cycle implementations. We are studying how it should be modified to fit a deeply pipelined microarchitecture and how this would impact performance.

DTM traces can be enlarged by allowing the inclusion of `LOAD` and `STORE` instructions, but with consequences upon implementation complexity. We are currently evaluating the cost-performance tradeoffs in the inclusion of memory access instructions to DTM traces.

The DTM technique has an interesting relationship with speculative execution control. It may avoid filling the pipeline with instructions from a mispredicted path, by immediately correcting the instruction fetch when a trace is reused. We are evaluating the performance improvements from such early path correction.

Currently, trace redundancy check is not performed if at least one of the input context registers has an invalid operand. Better performance from trace reuse could be achieved if valid register data were found more frequently. We are considering a hybrid of value prediction [11] and DTM to increase operand availability.

Acknowledgments

The authors wish to thank Edil Fernandes, Vinod Rebelo and Vitor Costa for their valuable contributions to this paper.

References

[1] A. Sodani, G. Sohi, *An Empirical Analysis of Instruction Repetition*, Proc. of the 8th ASPLOS Conference, 1998, pp. 35–45.

[2] D. F. Bacon, S. L. Graham, O. L. Sharp, *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys, Vol. 26, No. 2, Dec. 1994, pp. 345–420.

[3] D. Michie, *Memo Functions and Machine Learning*, Nature 218, 1968, pp. 19–22.

[4] A. Sodani, G. Sohi, *Dynamic Instruction Reuse*, Proc. of the 24th International Symposium on Computer Architecture, 1997, pp. 194–205.

[5] A. Sodani, G. Sohi, *Understanding the Differences Between Value Prediction and Instruction Reuse*, Proc. of the 31st International Symposium on Microarchitecture, 1998, pp. 205-215.

[6] J. Huang, D. Lilja, *Exploiting Basic Block Value Locality with Block Reuse*, Proc. of the 5th International Symposium on High-Performance Computer Architecture, 1999, pp. 106–115.

[7] V. E. F. Rebello, *NEUROCOM - Integrating Neurocomputing and Conventional Computing*, CNPq Project Technical Report, May 1997.

[8] B. Shriver, B. Smith, *The Anatomy of a High-Performance Microprocessor – A Systems Perspective*, IEEE Computer Society Press, 1998.

[9] A. Wolfe *et al.*, *Datapath Design for a VLIW Video Signal Processor*, Proc. of the 3rd Symposium on High-Performance Computer Architecture, 1997, pp. 24–35.

[10] A. Gonzalez, J. Tubella, C. Molina, *Trace-Level Reuse*, Proc. of the International Conference on Parallel Processing, 1999, pp. 30–37.

[11] M. H. Lipasti, J. P. Shen, *Exceeding the Dataflow Limit Via Value Prediction*, Proc. of the 29th International Symposium on Microarchitecture, 1996, pp. 226-237.