Evaluating DTM in a Superscalar Processor Architecture

Amarildo T. da Costa[⋆,†] Felipe M. G. França[⋆] Eliseu M. C. Filho[⋆]

[⋆]Dept. of Systems and Computer Engineering
COPPE/Federal University of Rio de Janeiro
P.O. Box 68511
21945-970 Rio de Janeiro, Brazil
{amarildo, felipe, eliseu}@cos.ufrj.br

[†]Dept. of Electrical Engineering
IME - Military Institute of Engineering
Pça Gal. Tibúrcio 80
22290-270 Rio de Janeiro, Brazil

Abstract

Dynamic Trace Memoization (DTM) is a reuse technique that employs memoization tables to skip the execution of *sequences* of redundant instructions. DTM thus extends the concept of instruction reuse to larger grained units and, contrary to other proposed reuse schemes, it is not constrained by architectural parameters nor code-level boundaries. For the benchmark programs in the SPECInt95 suite, evaluation results show that DTM improves performance by 5% to 21% with an average of 9.3%. For the largest common subset of the SPECInt95 benchmarks tested in two other previously proposed reuse mechanisms, DTM attains twice the average performance increase for configurations with similar storage capacities.

1 Introduction

Experimental studies [1, 2] demonstrate that *redundant instructions* represent a significant portion of the instructions executed by a program. 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 input data sets.

Some redundant computations can be removed by compiler transformations like common subexpression elimination and loop-invariant code motion [3]. However, given the restricted knowledge of the input data at compile-time, such optimizations are not effective in finding redundancies that manifest themselves only at run-time. For this reason, execution-time approaches are important in removing redundant computations.

Function Memoization [4] is an example of a run-time 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 hardware-based proposal was directly inspired by function memoization.

Redundancy elimination in hardware [5, 6, 7] is accomplished by caching the results of redundant instructions within the processor and reusing them on later occurrences of the instructions. This is called *dynamic instruction-level value reuse* [5] (the term *reuse* will be used throughout referring to this approach). Reuse decreases the dynamic instruction count, allows early resolution of data dependence chains [5], reduces branch resolution latency [6] and alleviates resource contention [6].

This paper presents the *Dynamic Trace Memoization* (DTM) technique [8], which employs *memoization tables* to capture and reuse *redundant traces*, i.e., sequences of redundant instructions. Some of the advantages of DTM are: (1) it potentially exploits more redundancy, as the traces cross basic block boundaries; (2) it is independent of the underlying microarchitecture parameters, since the trace length is not constrained by the dispatch width nor by instruction alignment within the dispatch window; and (3) it is expected to have a small impact on cycle time, given that reuse does not involve checking dependence chains.

Let us define the *actual reuse* as the fraction of the available redundancy which is in fact exploited by a given reuse mechanism. Experimental data indicate that DTM can deliver a significantly better actual reuse than the S_{n+d} [5, 6] and block reuse [7] mechanisms, which are also able to reuse sequences of instructions. For integer programs in the SPEC95 benchmark suite, DTM achieves an average actual reuse of 44%, compared to 19% for S_{n+d} and 22% for block reuse. For the subset of benchmark programs common the three studies, DTM introduces an average speedup of 10.2%, while S_{n+d} with the same store capacity and block reuse with twice the storage capacity provide average speedups of 4.4% and 3.4%, respectively.

The remainder of this paper is organized as follows. Section 2 provides a functional description of the DTM technique, while Section 3 shows how DTM could be incorporated into a typical superscalar microarchitecture. Section 4 reviews the related works. Section 5 presents and discusses 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 when it is only comprised of redundant instructions. The *DTM validity domain* is defined as the subset of

processor instructions that can be part of a trace. Of the instructions in the MIPS ISA, `LOAD`, `STORE` and `SYSCALL` instructions do not belong to the DTM validity domain. DTM does not reuse the values loaded or stored by memory access instructions, but it does reuse address calculations associated to those instructions.

The *input context* of a trace is defined as the set of operand values used by instructions in that trace, but which are produced by instructions outside the trace. An instance of a trace is redundant if its input context is identical to the input context of a previous instance of the trace. The *output context* of a trace is the set of results produced by instructions within that trace.

2.1 Construction of Redundant Traces

In the DTM technique, the construction of a redundant trace involves (1) determining whether each instruction is redundant and (2) gathering the trace's input and output contexts. Detecting redundant instructions means verifying whether the current instruction instance has the same operand values already seen by a previous instance. For just such purpose, DTM employs a structure called *global memoization table*, or *Memo_Table_G*.

Each *Memo_Table_G* entry corresponds to an instruction instance and has the format depicted in Figure 1. The field *pc* contains the instruction's address, the fields *sv1* and *sv2* hold the operand values and the field *res/targ* holds either the result of an arithmetic-logical instruction or the target address of a control transfer instruction. The bits *jmp* and *brc* identify the instruction as a jump or branch respectively, and bit *btaken* indicates whether the branch was taken or not.

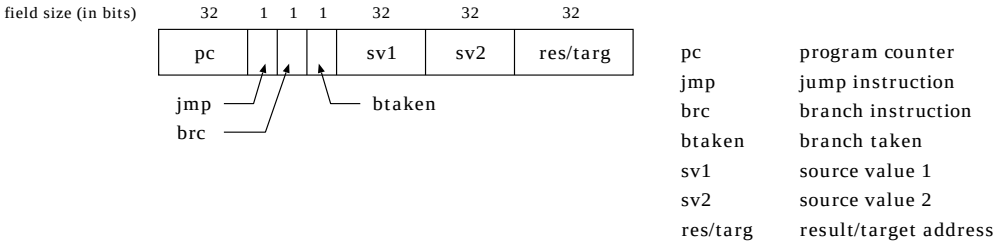


Figure 1: Structure of a *Memo_Table_G* entry.

DTM initially verifies whether each dynamic instruction belongs to the validity domain. If the instruction is invalid, then DTM labels the instruction as non-redundant and does not insert it into *Memo_Table_G*. Otherwise, DTM compares the instruction address and current operand values against the *pc*, *sv1* and *sv2* fields in the *Memo_Table_G* entries. If a match does not occur, then DTM labels the instruction as non-redundant, inserts it into a *Memo_Table_G* entry and fills the *pc*, *sv1* and *sv2* fields. If a match occurs, then DTM labels the instruction as redundant but does not insert it into *Memo_Table_G*. If the instruction is

found to be non-redundant, then DTM finishes any trace under construction. Otherwise, DTM either updates context information for the trace under construction or starts a new trace.

The structures employed by DTM to assemble and store trace information are shown in Figure 2. Both the *input context map* and *output context map* have a bit for each architectural register. An asserted bit in the input (output) context map indicates that the corresponding register contains a value which is part of the trace's input (output) context. DTM activates a bit of the input context map if (1) the respective register holds an operand value of an instruction within the trace and (2) the register was not last written to by an instruction within the trace. DTM activates a bit of the output context map if the respective register is the destination of an instruction. Upon setting a bit of the input (output) context map, DTM adds the respective register's contents to the trace's input (output) context.

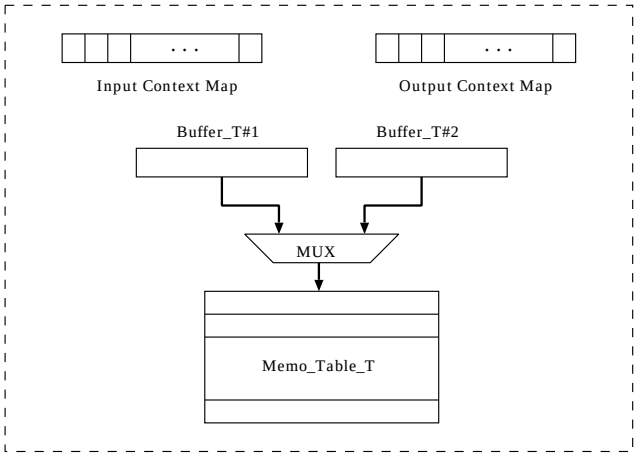
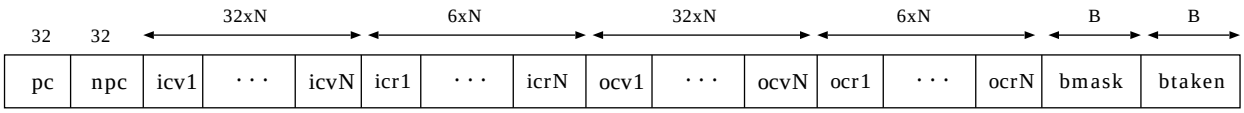


Figure 2: Structures for trace assembly and storage.

Buffers *Buffer_T#1* and *Buffer_T#2* hold information about the trace under construction, whereas data regarding constructed traces are kept in the *trace memoization table* *Memo_Table_T*. Once the construction of a trace is complete, DTM transfers trace data from a temporary buffer to a *Memo_Table_T* entry. Note that the construction of a new trace may begin at the same time as a previous trace is completed, provided a non-redundant instruction is found among the instructions being simultaneously analyzed. The alternate use of the two temporary buffers allows DTM to start constructing a new trace while it saves information pertaining to a previous trace. Traces could be incrementally constructed in *Memo_Table_T* entries, however the adoption of auxiliary buffers allows a simpler hardware structure for *Memo_Table_T*. Figure 3 depicts the format of the *Memo_Table_T* entries and temporary buffers.

The field *pc* stores the address of the initial trace instruction. Note that *Memo_Table_T* may have multiple



pc	program counter
npc	next program counter
icv1 ... icvN	input context values
icr1 ... icrN	input context registers (one validity bit per field)
ocv1 ... ocvN	output context values
ocr1 ... ocrN	output context registers (one validity bit per field)
bmask	branch flags
btaken	branch results

Figure 3: Format of temporary buffers and *Memo_Table_T* entries.

instances of the same trace (identical *pc*'s), but which differ in their input contexts. The field *npc* specifies the address of the next instruction to be executed in case the trace is reused, and it is filled with the address of the instruction following the last one in the trace (note that *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 (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 3, each register identifier field has an associated validity bit. Each bit in the *bmask* field indicates the presence of a branch instruction within the trace, while the corresponding bit in the *btaken* field indicates whether the branch was taken or not.

In Figure 3, 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. DTM finishes the construction of a trace 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.

2.2 Reusing Instructions and Traces

Concurrently with trace construction, DTM looks for redundant instructions and traces for reuse. The operations performed to detect redundant instructions were described in the previous subsection. In order to detect a redundant trace, DTM initially checks whether each dynamic instruction is the starting instruction of a previously seen trace, by comparing the instruction's address against the *pc* fields in *Memo_Table_T*. For those *Memo_Table_T* entries with matching *pc* fields, the current contents of the registers pointed by the valid

icr fields are compared against the input context values stored in the corresponding *icv* fields. The trace is redundant if the current input context (i.e., the set of register contents) matches a previous input context.

If a hit occurs only in *Memo_Table_G*, then the result in the appropriate *res/targ* field (Figure 3) is reused. In particular, if the redundant instruction is a either a jump or branch, then information in the *res/targ* and *btaken* fields is used to redirect instruction fetch and to update branch prediction state.

If a hit occurs in both memoization tables, DTM gives priority to trace reuse. In this case, the output context values in the *ocv* fields are written into the destination registers indicated by the valid *ocr* fields (Figure 3) and the address in the *npc* field is loaded into the program counter to skip the instructions covered by the trace. In addition, branch prediction state is updated with information in the *btaken* field.

3 A Microarchitecture with DTM

The previous section provided a conceptual, implementation-independent description of the DTM technique. This section shows how DTM could be added to an existing superscalar microarchitecture. To demonstrate the implementation feasibility of the DTM technique, we consider a substrate microarchitecture representative of those found in current superscalar processors. Except for a few differences[1], the microarchitecture depicted in Figure 4 is similar to that embodied in the AMD K6-III processor [9].

The *Instruction Fetch Stage* places instructions into the *Instruction Buffer*. The *Decode Stage* accesses instructions from that buffer and dispatches decoded instructions to the *Scheduler*. The Scheduler is comprised by two pipeline stages, the *Operand Fetch Stage* and the *Instruction Issue Stage*, and by the *Issue Buffer*. The Issue Buffer is logically organized as rows with w entries each, where w is the dispatch width. It provides both centralized reservation stations and reorder buffer functionality, by holding the results of executed instructions until they are committed. Note that each Issue Buffer entry acts as an implicit renaming register [9].

The Operand Fetch Stage reads operand values from the register file, updates validity information for the destination registers and inserts the instructions along with their valid operand values into the top row of the Issue Buffer. The Issue Stage selects instructions for execution according to operand availability. The *Execute Stage* contains the functional units. The *Commit Stage* updates the architectural state in program order. A given instruction is committed if the preceding instructions in the bottom row of the Issue Buffer have all been committed and if the instruction is not preceded by a mispredicted branch.

[1] The main difference being the order of the Operand Fetch and Instruction Issue stages in the pipeline.

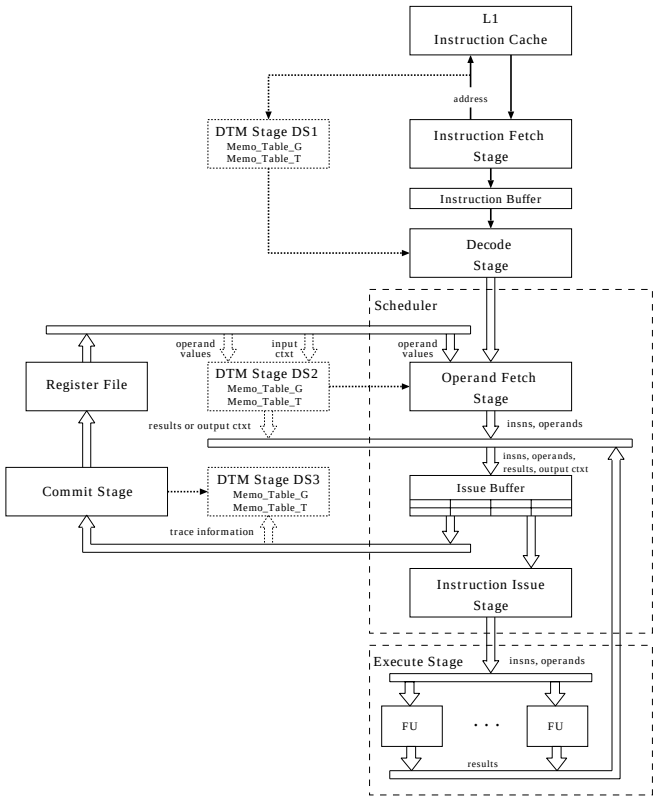


Figure 4: A typical superscalar microarchitecture (with DTM).

3.1 Adding the DTM Mechanism

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. The DTM stages and their datapaths appear in dotted lines in Figure 4 (the memo tables accessed by the DTM stages are indicated for each stage). A brief description of how the DTM stages operate is provided next. Operations in (1), (2) and (3) are related to trace construction, while operations in (4), (5) and (6) are in connection with instruction or trace reuse.

(1) Inserting instructions into Memo_Table_G. DS1 allocates a *Memo_Table_G* entry for each instruction fetched, fills the *pc* fields and forwards the indexes of the newly allocated entries to the Decode Stage. The Decode Stage attaches a *Memo_Table_G* index to the corresponding instruction before dispatching it to the Operand Fetch Stage. DS3 captures the operand values and result of each instruction read by the Commit Stage. If the instruction belongs to the DTM validity domain, then DS3 fills the remaining fields in the *Memo_Table_G* entry pointed by the index attached to the instruction. In the case where the instruction is either squashed or invalid, DS3 releases the appropriate *Memo_Table_G* entry. Note that DTM captures only instructions along *correct* execution paths.

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

(3) Updating trace information. DS3 captures the source register identifiers, the operand values, the destination register identifier and the result of each instruction committed by the Commit Stage. If the instruction is redundant, DS3 updates the context bitmaps and fills the appropriate fields in a temporary buffer (recall Figures 2 and 3). If the instruction is not redundant, then DS3 finishes trace construction by transferring trace information from the temporary buffer to a *Memo_Table_T* entry.

(4) Selecting candidate redundant instructions and traces. DS1 compares fetch addresses against the *pc* fields in both *Memo_Table_G* and *Memo_Table_T* and selects those entries with matching *pc*'s (the selected *Memo_Table_G* entries are common to (2) and (4)). If different fetch addresses match *Memo_Table_T pc* fields in the same cycle, then DS1 selects only the entries holding instances of the same trace (i.e., those with the same *pc* value).

(5) Identifying the instruction or trace as redundant from amongst the candidates. DS2 looks for redundant instructions in *Memo_Table_G* as described in (2) (redundant instructions are common for both (2) and (5)). In addition, DS2 reads the registers indicated by the valid *icr* fields in the previously selected *Memo_Table_T* entries and compares the operand values against the *icv* contents (from (4), note that corresponding *icr* values across the entries are identical).

(6) Reusing an instruction or a trace. If hits only occur in *Memo_Table_G*, then the Operand Fetch Stage inserts the redundant instruction(s) and result(s) (obtained from *Memo_Table_G*) into the Issue Buffer. If a hit in *Memo_Table_T* occurs, then the Operand Fetch Stage inserts 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.

3.2 Implementation Issues

It is important to realize that the proposed DTM mechanism does not add stages to the pipeline depth and therefore the effective reuse latency is null. The introduction of the DTM stages is not expected to impact the cycle time. The associative searches performed by DS1 are overlapped by the access to the instruction cache,

and the operations performed by stages DS2 and DS3 are hidden by operand fetch and instruction commit.

The major technical challenge brought forward by DTM is the requirement of the additional read ports in the register file, needed to allow simultaneous accesses by DS2 and the Operand Fetch Stage. This should not represent a real obstacle to a DTM implementation, for two reasons. First, DTM does not require a large number of additional read ports: we have estimated no more than four additional read ports (according to our measurements) are necessary. Second, modern fabrication technologies make it feasible to implement multiported register files. Wolfe *et al.* [10] show that the delay of a register file for a VLIW video signal processor is not severely affected by the number of ports, although area can increase by a factor of ten when the number of ports varies from three to twelve. Using a 0.25 μm, 4-metal layer technology, they quote a 12-port, 128×16-bit register file as occupying 3.0 mm^2 and operating at 650 MHz. Large multiported register files are present even in older designs. For example, the SPARC64 [11] has a 14-port, 116×64-bit register file. Extending the microarchitecture of the AMD K6-III, which has a register file with nine read ports [9], with DTM would require a register file with thirteen read ports.

3.3 Handling Branch Instructions

Prediction state should be updated whenever a branch instruction is reused in order to obtain the same effect were the branch executed. In single-level, BHT-based predictors [12] and two-level predictors with per-address or per-set histories [13], the selection of the prediction bits requires an index or tag derived from the branch instruction's address. On the other hand, two-level predictors with global histories [13, 14] do not require address information for state update.

Let us first consider predictors that require instruction address information. When a branch instruction is reused individually, its address is available from the *pc* field in the corresponding *Memo_Table_G* entry. The selected state bits are then updated with the branch outcome indicated by the *btaken* field. However, for branch instructions which are reused as part of a trace, *Memo_Table_T* should store the multiple branch addresses. New fields could be added to the *Memo_Table_T* entries to hold the addresses of the branch instructions within the trace, but this significantly increases the implementation cost. Alternatively, each *ocv* field (Figure 3) could store either the result of an arithmetic/logical instruction or the branch instruction address, with the type of the information stored being indicated by the corresponding *bmask* bit. Due to the sharing of *ocv* fields, the modified DTM mechanism might require a larger context size N in order to preserve the same reuse levels of a DTM mechanism which does not share the output context fields. However, the

additional cost of this alternative is less than that incurred by explicit address fields.

In the case of branch predictors that do not need branch address information, the described DTM mechanism supports prediction state updates without any modification. Therefore, it is fair to say that, cost-wise, DTM is more suited to microarchitectures employing two-level, global history branch predictors. This does not represent a significant disadvantage: as shown in [13], the prediction accuracy achieved by a 32K bits global history scheme is 1% inferior to that delivered by a 32K bits per-address scheme.

4 Related Works

In [5], Sodani and Sohi introduce the concept of *dynamic instruction reuse* and propose three reuse schemes named S_v, S_n and S_{n+d}. All are based on a structure called *Reuse Buffer* (*RB*), which saves operand information and results to allow the execution of redundant instructions to be skipped. The schemes S_v and S_n reuse single instructions, however S_{n+d} is able to reuse chains of interdependent instructions. In [7], Huang and Lilja introduce the notion of *basic block value locality* and present the *block reuse* technique. The 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 a number of important aspects. The main differences to S_{n+d} are:

(1) In order to determine whether an instruction chain can be reused, S_{n+d} must verify, for each instruction, if the dependence relationships remained unaltered since the last execution of the instruction [5]. This requires sequential select, read and comparison operations across two tables (the *RB* and the *Register Source Table*, *RST*) within a single cycle. DTM employs a parallel process to decide whether a trace can be reused, by associatively checking the input contexts of previous trace instances. The associative searches are hidden by other associative operations already performed in the processor, namely, cache accesses. Also, the delay of the associative searches can be reduced via optimized purely-associative designs [15] or by using set-associative tables.

(2) In S_{n+d}, dependence relationships are evaluated only for those instructions within the same dispatch cycle [5]. 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 [16];

(3) In S_{n+d}, whether or not a given instruction chain is completely reused in a single cycle depends on the dispatch slot occupied by its first instruction. For 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;

(4) For a chain to be entirely reused, S_{n+d} must keep all the instructions that form the chain in the *RB* [5]. 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 need to retain the instructions covered by traces. Moreover, the capture of redundant instructions does not interfere with trace reuse, as these two processes manipulate distinct memo tables;

The main differences between DTM and block reuse are:

(1) Block reuse exploits redundancy within basic block boundaries, whereas DTM is not constrained by code-level boundaries. As happens with global instruction scheduling and control speculation, crossing the boundaries of basic blocks results in better performance potential;

(2) Block reuse requires compiler assistance in order to detect and mark dead definitions from basic blocks [7], 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;

(3) In block reuse, the *BHB* stores the input and output values for a basic block from its previous four executions [7]. 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 (with different input contexts) and by releasing entries not frequently used. This results in a better utilization of hardware resources.

In [17], Gonzalez *et al.* study the potential of value reuse at the trace level. A memory structure to store trace information, the *Reuse Trace Memory* (*RTM*), is described. However, some issues are not fully addressed, for example: how to incorporate the *RTM* into a real microarchitecture, how to keep the *RTM* consistent with data memory and how to handle branch instructions. Assuming an ideal base machine, an infinite *RTM*, maximum-length traces and both infinite and 256-entry instruction windows, the upper bounds for the speedup that can be achieved with trace-level reuse are shown. For *RTM* sizes ranging from 128 KBytes to 64 MBytes and assuming that the reuse mechanism can read and write 16 register/memory values

per cycle, the percentage of reused instructions and the average trace sizes are also quantified. No speedup figures are given in this latter case. In this work we approach trace-level reuse from another perspective, by not only proposing a different scheme, but also providing a more practical evaluation.

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 penalties related to keeping the memo tables consistent with respect to the data memory. To maintain consistency, it would be necessary to detect `STORE` instructions writing into memory locations referenced by `LOAD` instructions in the memo tables, in order to either update or invalidate the appropriate table entries. This would require larger memo table entries, extra access ports and additional associative comparators, with consequences on the datapath complexity, cost and performance (cycle time). Moreover, invalidations could unnecessarily throw away multiple traces, in the case a `STORE` re-writes the same value already stored in the memory. Of course, it is possible to check such situations and avoid the invalidation but, again, this requires extra hardware. As the evaluation results in the next section will demonstrate, the good features of DTM compensate for the exclusion of memory access instructions.

5 Experimental Evaluation

We have employed the *SimpleScalar Tool Set, Version 2.0* [18] to evaluate the Dynamic Trace Memoization technique. SimpleScalar's out-of-order timing simulator, `sim-outorder`, was modified to include the three DTM stages (the microarchitecture simulated by `sim-outorder` is similar to that shown in Figure 4). Table I summarizes the configuration of the base out-of-order simulator adopted in the experiments.

Table I: Characteristics of the base out-of-order simulator.

instruction fetch	4 instructions per cycle, one taken branch per cycle, cache accesses do not cross line boundaries
branch predictor	2048-entry bimodal
instruction issue and execution	4 instructions per cycle, 16-entry reorder buffer, 8-entry load/store queue, loads processed when all previous store addresses are known, store forwarding
registers	32 integer, 32 floating point
functional units	4 integer ALUs, 2 load/store units, one integer mult/div 4 FP adders, one FP mult/div
latencies	integer ALU 1, memory access 1, i-mult 3, i-div 20, fp-adder 2, fp-mult 4, fp-div 12
i-cache and d-cache	16 KBytes, 2-way, 32-byte line, 6-cycle miss latency

The SPEC95 integer programs were used 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 ran to completion except for `perl` and `vortex`, which executed up to 300 million instructions.

The next subsection presents the performance gains and the actual reuse attained by DTM. The effect on performance is 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. Reused address calculations in memory access instructions are taken into account when counting N_r. The memory access is counted as a single instruction within the dynamic instruction count N_t.

5.1 Performance Gains and Actual Reuse

Figure 5 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. 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`). As Figure 5 shows, 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.

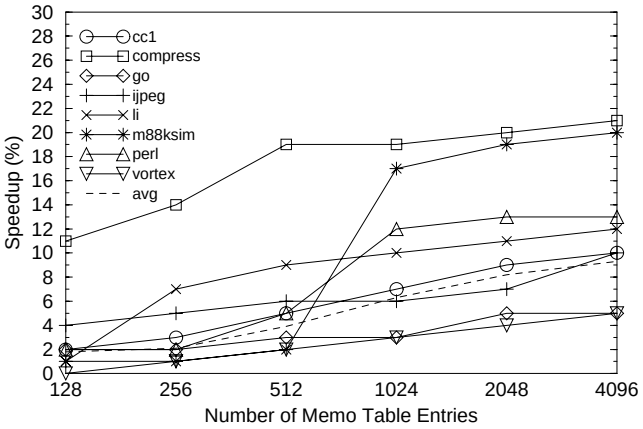


Figure 5: Speedup provided by DTM.

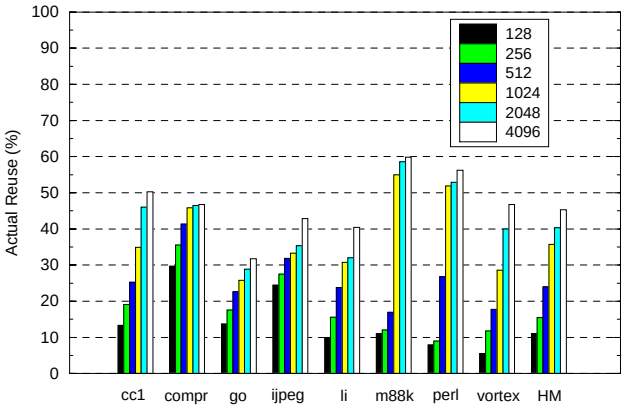


Figure 6: Actual reuse provided by DTM.

Figure 6 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

Figures 7 and 8 show, respectively, the distribution of input context sizes and traces sizes as a percentage of the number of traces actually reused. From Figure 7, 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 (this justifies why we have assumed *Memo_Table_T* configurations with $N = 4$ in our experiments). Note that `go` and `vortex`, for which DTM performed worst, are the benchmarks with the highest frequency of zero-operand input contexts. Regarding Figure 8, 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).

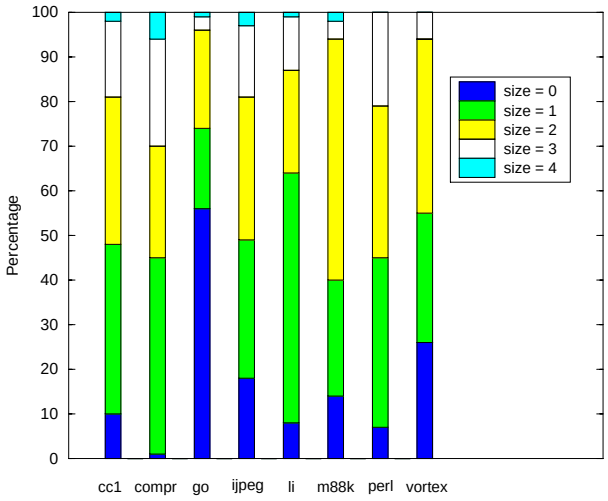


Figure 7: Input context size.

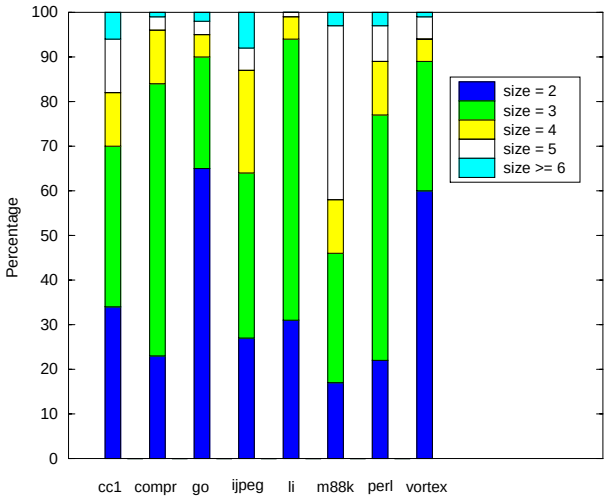


Figure 8: Trace size.

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, configurations of DTM and S_{n+d} with the same storage capacity are considered. In the augmented S_{n+d} described in [6], 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 784 Kbits. Each *Memo_Table_G* entry has 131 bits (Figure 1) and, assuming $N = B =$

4, each *Memo_Table_T* entry has 376 bits (Figure 3); 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 storage capacity among those for which experimental data provided in [7]. 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 storage capacity of the DTM and S_{n+d} configurations.

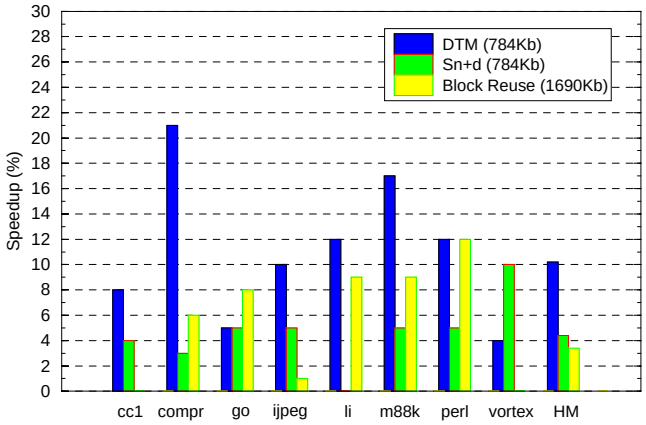


Figure 9: Speedup comparison.

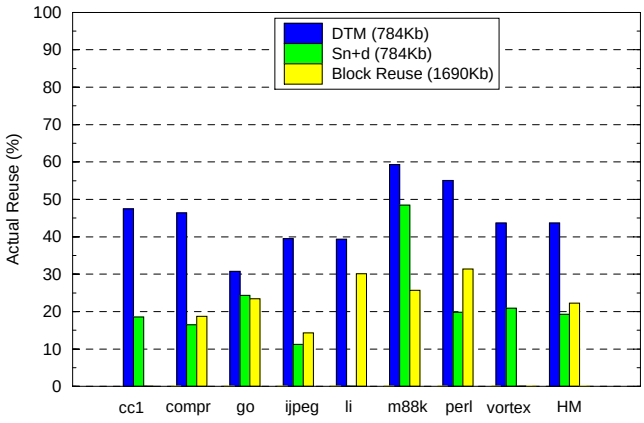


Figure 10: Actual reuse comparison.

Figure 9 shows the percentage speedups provided by DTM, S_{n+d} and block reuse. Speedups for S_{n+d} and block reuse are given in [6] and [7], respectively (speedups are not reported for `li` in [6] nor for `cc1` and `vortex` in [7]). These two other studies also employed the SimpleScalar simulator with similar configurations. 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 [6] and [7], 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.

Explanations regarding the exception cases are in order (given that block reuse has more than twice the storage capacity of DTM and S_{n+d} and that storage capacity is a major factor to performance, we focus only on the behavior relative to S_{n+d}). 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: in fact, additional measurements (not shown here) indicate that, for 75% of the reused traces, the critical path length within those traces is just one instruction. This means that most of the instructions forming these traces could be executed in parallel by the substrate microarchitecture (without DTM), provided that the necessary resources were available. 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, which is the highest occurrence of this instruction class amongst the benchmarks considered here. 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 10 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 reported in [6] and [7], 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 [17], the average (geometric mean) number of redundant instructions executed by the SPEC95 integer benchmarks accounts for 83% of the dynamic instructions.

From Figure 9, 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 10, 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. Similar reasoning may also explain why the performance of S_{n+d} is not directly proportional to its actual reuse. For example, speedups of S_{n+d} for `go` and `vortex` are 5% and 10% respectively (Figure 9), however the actual reuse for these two programs is 24% and 21%, respectively (Figure 10). Similar behavior can be observed when comparing DTM speedup and actual reuse (Figures 5 and 6).

5.4 Cost-effectiveness of DTM

The previous results are based on configurations in which *Memo_Table_G* and *Memo_Table_T* have the same number of entries. The following set of results explore the design space of DTM in more detail, by presenting the effect of each memo table size on the performance.

Figure 11 shows the percentage speedup as the size of *Memo_Table_G* increases from 128 to 4096 entries with the size of *Memo_Table_T* kept fixed at a large value of 4096 entries. Performance gains are more noticeable as the size of *Memo_Table_G* increases up to 1024 entries. Although improvements are still observable beyond that size, they are less significant and may not be worth the incurred additional hardware cost.

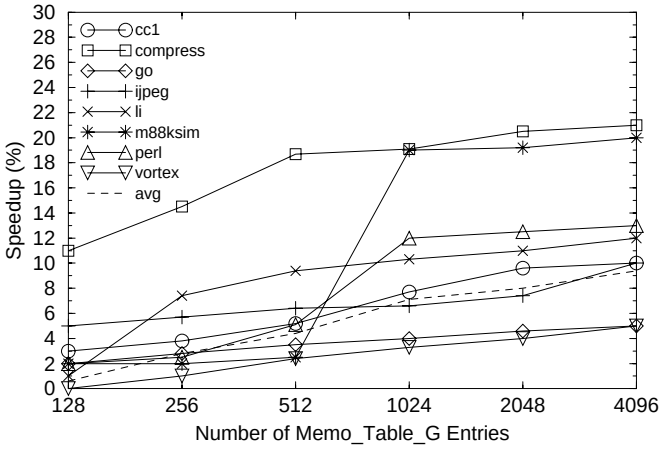


Figure 11: Effect of *Memo_Table_G* size.

Observe that the plot in Figure 11 resembles the one depicted in Figure 5. This suggests that the ability to cache redundant instructions, which is determined by the size of *Memo_Table_G*, is essential to the performance of DTM. This behavior was expected, as both the number of traces constructed and the trace sizes depend directly on the hit rate (and thereby on the size) of *Memo_Table_G*.

Figure 12 shows the speedup as the size of *Memo_Table_T* increases from 128 to 4096 entries. In Figure 12(a) the size of *Memo_Table_G* is fixed at 4096 entries, whereas in Figure 12(b) the size of *Memo_Table_G* is 1024 entries (the latter size probably represents the best cost-effective *Memo_Table_G* configuration, see Figure 11). The first plot shows speedups varying from 5.4% to 9.3% with speedups from 4.7% to 7.2% being observed in the second. In both cases, *Memo_Table_T* sizes greater than 1024 entries provide additional performance improvements of at most 1.2%.

Given the speedup sensitiveness to *Memo_Table_G* size observed in Figure 11 and the apparently less important effect of *Memo_Table_T* size shown in Figure 12, one might argue that the performance of DTM

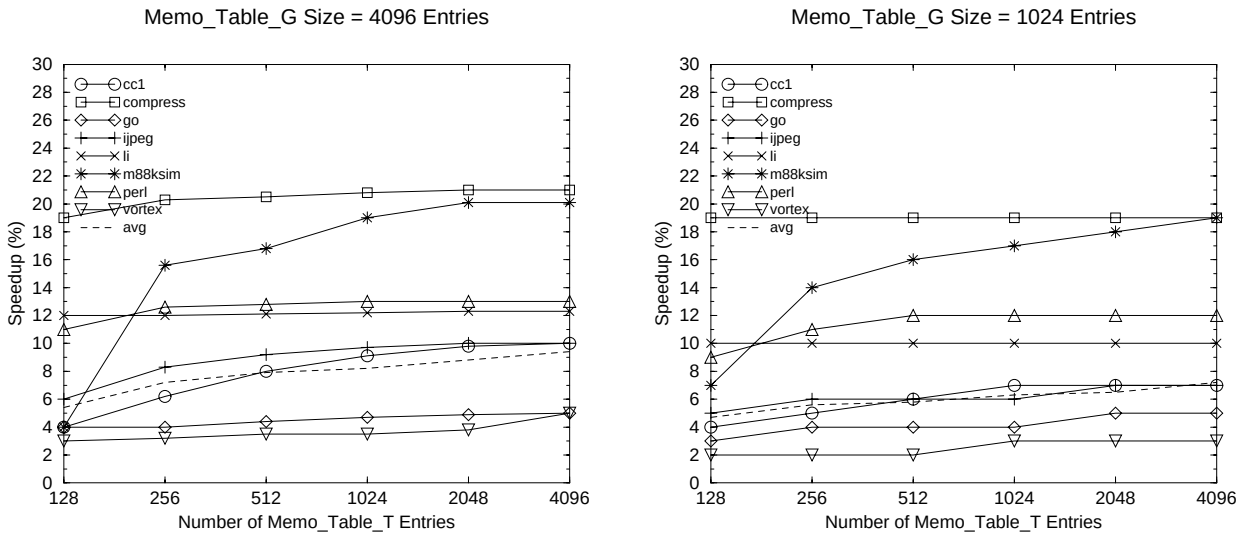


Figure 12: Effect of *Memo_Table_T* size.

comes mainly from individual instruction reuse. In order to demonstrate the importance of trace reuse, two sets of experiments were devised. Let us assume a total storage capacity of 784 Kbits, the same for a 4096-entry *RB*. In the first set of experiments, this storage capacity was entirely allocated to *Memo_Table_G*, i.e., *Memo_Table_T* was effectively removed. In the second set, a 512-entry *Memo_Table_T* was re-introduced and the remaining storage capacity was allocated to *Memo_Table_G*. The speedups measured in the two sets of experiments are shown in Figure 13. In the figure, MTG refers to the configuration with *Memo_Table_G* only and DTM refers to the full DTM mechanism.

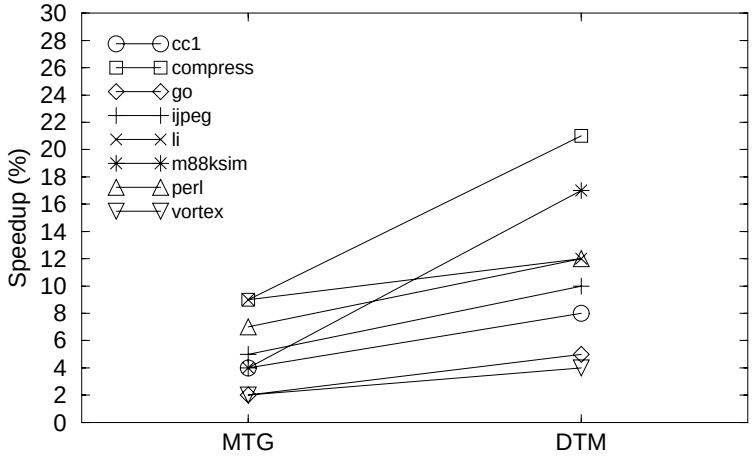


Figure 13: Effect of trace level reuse.

The performance improvements observed between the two cases emphasize the importance of reusing complete sequences of instructions, in addition to individual instructions. Note that configuration MTG is

equivalent to the S_v scheme introduced in [5], which reuses only individual instructions.

6 Conclusions and Future Works

This work introduced Dynamic Trace Memoization (DTM) as a technique to reuse sequences of redundant instructions. Compared to two other reuse schemes, DTM exploits redundancy across basic blocks boundaries, does not require dependence chain checking, is independent of architectural parameters and may consume less on-chip storage capacity. DTM has a feasible hardware implementation in pipeline stages which overlap the operation of the existing microarchitecture's stages.

For the benchmark programs considered in the experimental evaluation, DTM produced speedups from 5% up to 21%, with an average performance increase of 9.3%. DTM outperformed the other reuse schemes in most benchmark programs, with its average speedup being twice the average speedup of S_{n+d}. It is important to notice that DTM achieved better performance than S_{n+d} for a configuration with the same storage capacity, and better performance than the block reuse technique with more than twice the storage capacity.

With respect to the ability to exploit redundant instructions, DTM managed to reuse between 31% and 59% of the instructions executed in the benchmark programs. The average actual reuse was 44%, against an average actual reuse of 22% attained by S_{n+d} for a configuration with the same storage capacity. DTM has a better efficiency in exploiting instruction redundancy even without reusing memory access instructions. Assuming that on-chip storage area dominates implementation cost, the results presented here suggest that DTM has better cost-effectiveness characteristics than S_{n+d} and block reuse.

Two important aspects of DTM are under investigation. The first is in connection with speculation control. Recall that instruction fetch is redirected to the destination indicated by *npc* whenever a trace is reused (Section 2). If the branch predictor makes a wrong prediction in the same cycle that a trace is reused, DTM avoids filling the pipeline with instructions from the wrong path by immediately correcting the instruction fetch. Preliminary results indicate that such *early speculation correction* can introduce appreciable performance improvements.

The second issue concerns operand availability. Recall that, in order to detect whether a trace is redundant or not, the trace's input context values are compared to the current register contents. The greater the input context size, the higher the probability that the redundancy check cannot be performed due to invalid register data. Preliminary evaluations have shown that significantly better performance could be achieved if valid register data can be found more frequently. In order to get closer to this desirable case (i.e., register operands

being available most of the time), a Value Prediction Table [19] could be embedded into *Memo_Table_T*. The effectiveness of this hybrid of value prediction and trace reuse is being studied.

References

[1] Y. Sazeides, J. E. Smith, *The Predictability of Data Values*, Proc. of the 30th International Symposium on Microarchitecture, 1997, pp. 248–258.

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

[3] 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.

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

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

[6] 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.

[7] 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.

[8] V. E. F. Rebello, *NEUROCOM - Integrating Neurocomputing and Conventional Computing*, ProTem II-CC, CNPq, Brazil, Project Technical Report, May 1997.

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

[10] 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.

[11] T. Williams, N. Patkar, G. Shen, *SPARC64: A 64-b 64-Active Instruction Out-of-Order-Execution Processor*, IEEE Journal of Solid State Circuits, Vol. 30, No. 11, Nov. 1995, pp. 1215–1226.

[12] D. Patterson, J. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan-Kaufmann, 1997.

[13] T.-Y Yeh, Y. Patt, *A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History*, Proc. of the International Symposium on Computer Architecture, 1993, pp. 257–267.

[14] S. McFarling, *Combining Branch Predictors*, Technical Note TN-36, DEC Western Research Laboratory, June 1993.

[15] C. Zhang, X. Zhang, Y. Yan, *Two Fast and High-Associativity Cache Schemes*, IEEE Micro, Vol. 17, No. 5, Sept./Oct. 1997, pp. 40–50.

[16] IBM and Motorola, *PowerPC 750 RISC Microprocessor Technical Summary*, Order Number MPC750/D, 1997 (available at http://www.chips.ibm.com/products/ppc/).

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

[18] D. Burger, T. Austin, S. Bennett, *The SimpleScalar Tool Set, Version 2.0.* Technical Report 1342, Computer Science Department, University of Winsconsin.

[19] 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.