Exploiting Reuse with Dynamic Trace Memoization: Evaluating Architectural Issues

Amarildo T. da Costa[1 2], Felipe M. G. França[2], Eliseu M. C. Filho[2]

[1] Dept. of Electrical Engineering
IME - Military Institute of Engineering
Pça Gal. Tibúrcio 80
22290-270 Rio de Janeiro, Brazil
[2] Dept. of Systems and Computer Engineering
COPPE/Federal University of Rio de Janeiro
P.O. Box 68511
21945-970 Rio de Janeiro, Brazil
Comments and suggestions to: {amarildo,felipe,eliseu}@cos.ufrj.br

Abstract—

Employing memoization tables to skip the execution of dynamic sequences of redundant instructions, our Dynamic Trace Memoization (DTM) mechanism extends the concept of instruction reuse to larger grain units.

This work evaluates three critical architectural issues concerning the feasability of DTM:

(i) The cost-efectiveness of trace level reuse — the paper shows that a balance between the sizes of the single instruction memoization table and the trace memoization table produces higher speedups and does not need a higher number of read ports in the trace memo table;

(ii) Register file pressure — for the SPECInt95 and SPECFp95 benchmark suites, DTM requires no extra read ports;

(iii) Floating-point apparatus — in contrast with the speedup of 8.4% obtained by the SPECInt95, the SPECFp95 presented a speedup of 7% considering a DTM mechanism which ignores the floating-point operations.

Keywords— **Trace Reuse, Memoization, Instruction Reuse, Superscalar Processor**

I. Introduction

Redundant instructions represent a significant fraction of the instructions executed by a program [1, 2]. Redundant instructions are dynamic instances of the same static instructions which execute with the same operand values, thereby producing 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.

Redundancy elimination in hardware [3, 4, 5] 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 reuse* [3] (the term *reuse* will be used throughout referring to this approach). The main effects of reuse are: decreased dynamic instruction count, early resolution of data dependence chains [3], reduced branch resolution latency [4] and less resource contention [4].

This paper evaluate the *Dynamic Trace Memoization* (DTM) technique [7, 8] considering important architectural issues such as; (i) the cost-effectiveness of trace level reuse, i.e., a possible implementation balance between single instruction and trace *memoization tables* [9] (used to capture and reuse *redundant traces*, i.e., dynamic sequences of redundant instructions) producing higher speedups without infering in a higher number of read ports in the trace memo table; (ii) the pressure in the register file, i.e., it is shown that DTM requires no extra read ports when considering both SPECInt95 and SPECFp95 benchmark suites; (iii) floating-point aparatus - in contrast to the 4% to 21% (8.4% HM) speedups obtained upon running SPECInt95, speedups from 4% to 12% (7% HM) were observed upon running the SPECFp95 benchmark suite considering a DTM mechanism having no dedicated floating-point treatment. A comparison between DTM and other reuse techniques is also provided.

The following describes how the remainder of this paper is organized. Section II provides a functional description of the DTM technique, while Section III shows how DTM could be incorporated into a typical superscalar microarchitecture. Section IV presents a summary of related works. Section V presents and discusses the experimental evaluation. Section VI summarizes the main conclusions and future works.

II. Dynamic Trace Memoization

A *trace* is a dynamic sequence of instructions executed by a program. A trace is redundant when it is only comprised by redundant instructions. The *DTM validity domain* is the subset of processor instructions that can be part of a trace. Of the instructions in the MIPS ISA, only `LOAD`, `STORE`, `FLOATING-POINT` 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 effective address calculations). The *input context* of a trace is 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. We next explain how redundant traces are constructed and reused.

A. Construction of Redundant Traces

The construction of a redundant trace involves verifying whether the current instruction instance has the same operand values already seen by a previous instance. For such, DTM employs a *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 *btk* indicates whether the branch was taken or not.

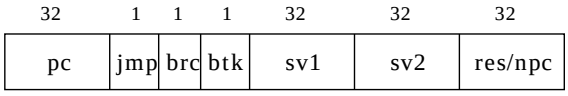

Fig. 1. Structure of a *Memo_Table_G* entry.

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 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.

Figure 2 shows the structures employed by DTM to assemble and store trace information. 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 the following two conditions are true: the respective register holds an operand value of an instruction within the trace; and 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.

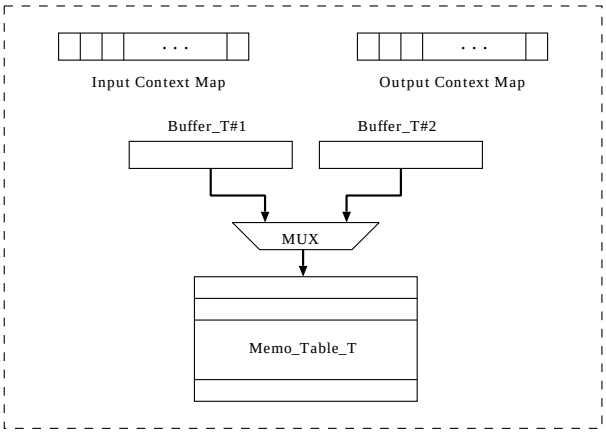

Fig. 2. Structures for trace assembly and storage.

Buffers *Buffer_T#1* and *Buffer_T#2* keep 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.

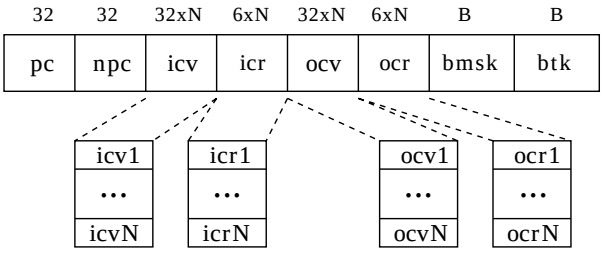

Fig. 3. Format of temporary buffers and *Memo_Table_T* entries.

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. The field *npc* specifies the address of the next instruction to be executed in case the trace is reused; it is filled with the address of the instruction following the last one in 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 *bmsk* field indicates the presence of a branch instruction within the trace, while the corresponding bit in the *btk* field indicates whether the branch was taken or not.

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.

B. Reusing Instructions and Traces

Concurrently with trace construction, DTM looks for redundant instructions and traces for reuse. 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 1) is reused. In particular, if the redundant instruction is a either a jump or branch, then information in the *res/targ* and *btk* 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 *btk* field.

III. A Microarchitecture with DTM

Let us consider a substrate microarchitecture representative of those found in current superscalar processors. Except for a few differences the microarchitecture depicted in Figure 4 is similar to that embodied in the AMD K6-III [10].

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 latter 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. The Issue Buffer provides both centralized reservation stations and reorder buffer functionality.

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.

Assuming this substrate microarchitecture, the DTM mechanism is organized into three pipeline stages, *DS1*, *DS2* and *DS3*, which operate in parallel with the Instruc-

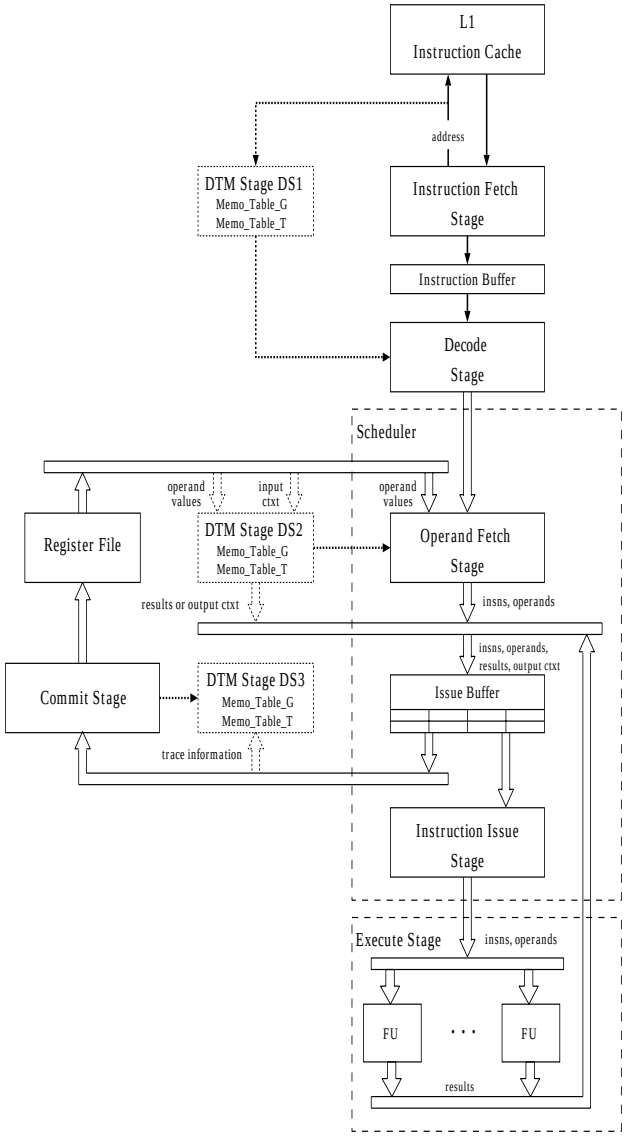

Fig. 4. A typical superscalar microarchitecture (with DTM).

tion 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 related to trace construction:

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.

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.

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. 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.

Operations related to trace reuse:

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. 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).

DS2 looks for redundant instructions in *Memo_Table_G*, as previously described. In addition, DS2 reads the registers indicated by the valid *icr* fields in the previously se-

lected *Memo_Table_T* entries and compares the operand values against the *icv* contents.

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.

IV. Related Works

In [3], 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 [5], 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.

In [11], 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.

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.

V. Experimental Evaluation

We have employed the *SimpleScalar Tool Set, Version 2.0* [12] 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).

The SPEC95 integer and floating-point programs were used as the benchmark programs. They were compiled by using the C compiler provided by 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 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.

A. 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 [4], 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. From the experimental results presented in [7], a configuration of DTM having a 512-entry *Memo_Table_T* and a 4672-entry *Memo_Table_G* (that also requires 784 Kbits of storage capacity) is chosen. 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 [5]. 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.

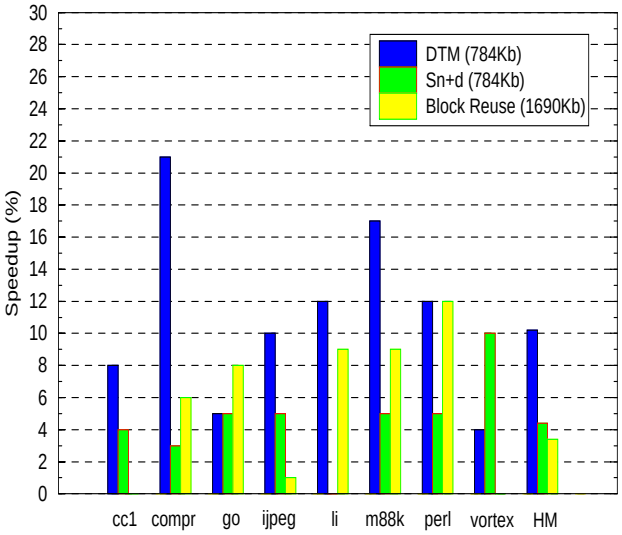

Fig. 5. Speedup comparison.

Figure 5 shows the percentage speedups provided by DTM, S_{n+d} and block reuse. Speedups for S_{n+d} and block reuse are given in [4] and [5], respectively (speedups are not reported for `li` in [4] nor for `cc1` and `vortex` in [5]). 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 [4] and [5], 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.

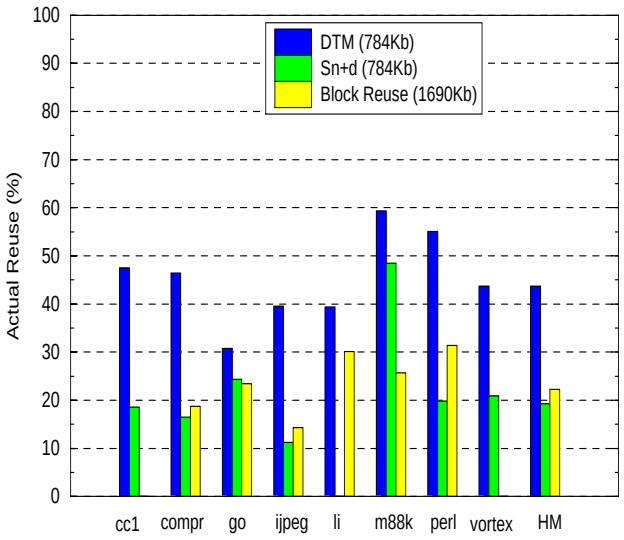

Fig. 6. Actual reuse comparison.

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 [7, 8], 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 criti-

cal 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 6 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 [4] and [5], 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 [11], the average (geometric mean) number of redundant instructions executed by the SPEC95 integer benchmarks accounts for 83% of the dynamic instructions.

From Figure 5, 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 6, 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 5), however the actual reuse for these two programs is 24% and 21%, respectively (Figure 6). Similar behavior can be observed when comparing DTM speedup and actual reuse [7, 8].

B. Architectural Issues

The previous results are based on a balanced configuration of DTM in which a 4672-entry *Memo_Table_G* and a 512-entry *Memo_Table_T* are assumed. The following set of results are obtained by assuming the same balanced configuration in both SPECInt95 and SPECFp95 benchmark programs, i.e., no modification on the domain set of reusable instructions is made and no floating-point registers are considered by the DTM mechanism.

A comparison between the balanced DTM configuration and a single-instruction-only DTM configuration where the whole 784 Kbits are dedicated only to *Memo_Table_G* (i.e., *Memo_Table_T* is not used and no trace reuse is made) is offered in Figures 7, 8, 9 and 10 for both SPECInt95 and SPECFp95 benchmark suites. A first interesting observation lies on the importance of trace reuse: although aproximately a 20% increase on the amount of reuse is observed when trace reuse is made (Figures 7 and 8), almost 50% of the whole performance improvement given by DTM is due to trace reuse in both cases (Figures 9 and 10). 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 [3], which reuses only individual instructions.

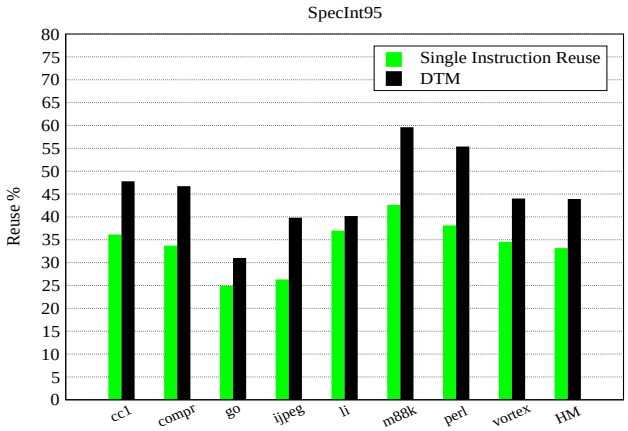

Fig. 7. Effect of trace level reuse on Total Reuse.

A second important observation that can be made from

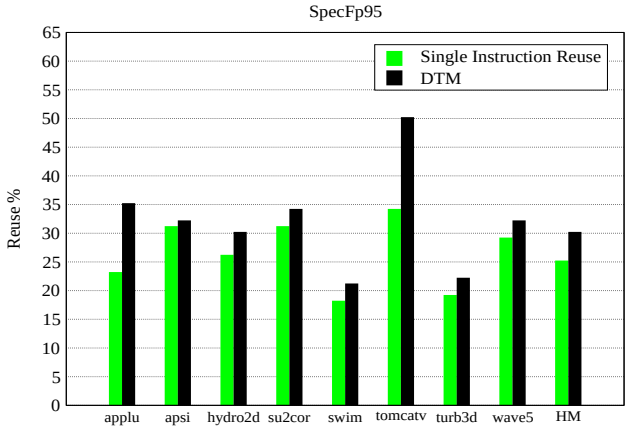

Fig. 8. Effect of trace level reuse on Total Reuse.

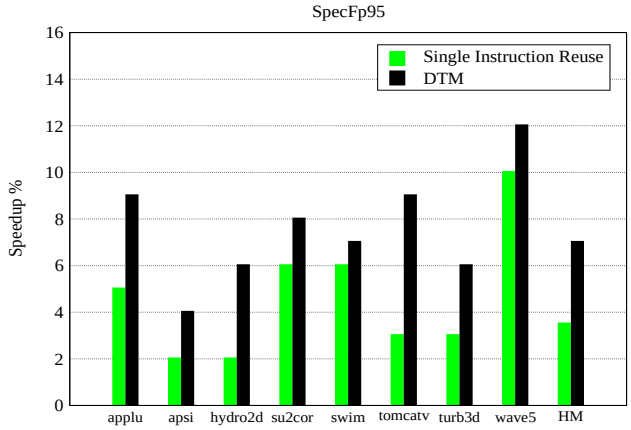

Fig. 10. Effect of trace level reuse on Speedup.

Figure 8 and 10 is the noticeable total reuse and speedups figures obtained from running SPECFp95 programs on the balanced DTM configuration where no special floating-point apparatus were considered. Reuse amounts from 21% (`swim`) to 50% (`tomcatv`) were observed on the definition of an average of 30% HM. Speedups from 4% (`apsi`) to 12% (`wave5`), average of 7% HM, are considerably good results.

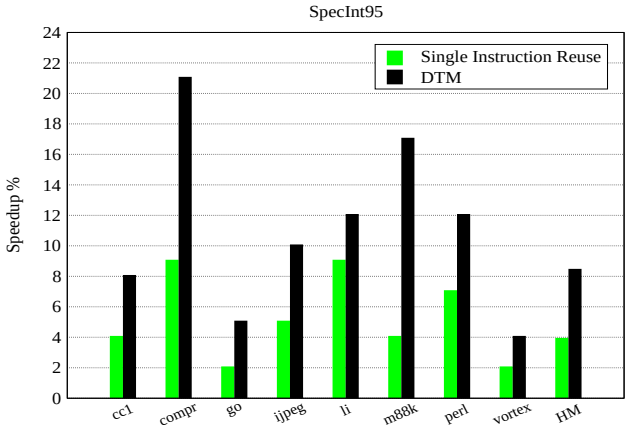

Fig. 9. Effect of trace level reuse on Speedup.

Another important consideration refers to a possible increase on the pressure on the register file of the substrate microarchitecture, i.e., the possibility of simultaneous reuse of traces and provision of operands for ongoing execution of instructions could mean that extra read ports should be required by the register file. Notice that no extra write ports are ever needed since the output context of a reusable trace is given to the register file by including instructions (marked as executed) in the instruction window/reorder buffer that will load each output context value into the corresponding registers when committed. Figures 11 and 12 presents, considering SPECInt95 and SPECFp95 programs, respectively, distributions of simultaneous number of read ports used in the register file when running the substrate microarchitecture with DTM. Although it could be shown that there are hypothetical situations where many extra ports would be required, it was found that DTM require no extra read ports for the whole SPEC95 benchmark suite.

It was assumed that such troublesome hypothetical situations are very unlikely to occur and that, considering that 99.6% of all reusable traces have up to four (4) operands as input context [7, 8], the addition of four (4) extra read ports to the register file would be reasonable as an implementation tradeoff. Modern fabrication technologies make it feasible to implement multiported register files. Wolfe *et al.* [14] 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 [15] 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 [10], with DTM

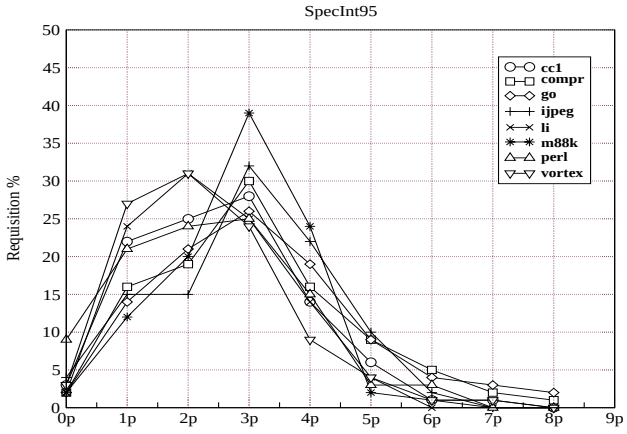

Fig. 11. Register file pressure for SPECInt95.

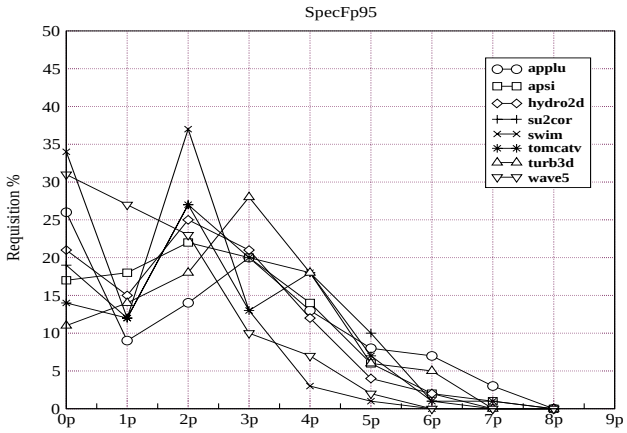

Fig. 12. Register file pressure for SPECFp95.

Another architectural issue of concern is over the number of read and write ports that should be present at *Memo_Table_T*. As each reusable trace is incrementally mounted in *Memo_Table_T* from committed instructions (see Section II, Subsection A), just one write port is needed since an interruption in a trace construction would mean that, in the worst case, another trace construction would follow. Figures 13 and 14 presents the distributions of the number of simultaneous accesses to *Memo_Table_T* triggered by DS2 upon running SPECInt95 and SPECFp95 programs, respectively.

Each time an I-Cache line is loaded into the Fetch queue, up to four (4) instructions can induce the DS1 stage to start simultaneous associative accesses to

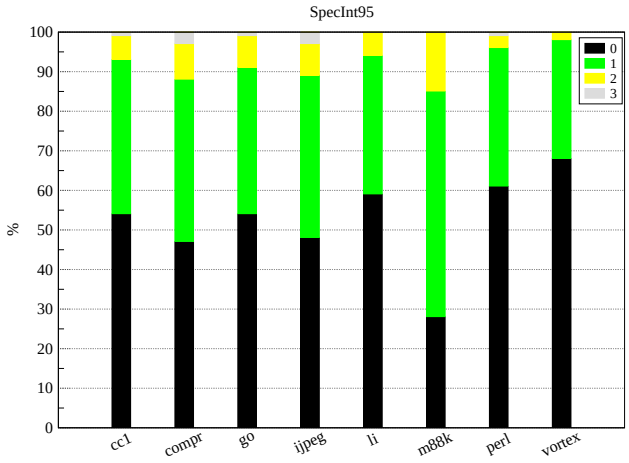

Fig. 13. Number of matching traces per clock cycle.

Memo_Table_T. However, the two distributions found show that the DS2 stage implies that, in practical terms, up to two (2) simultaneous accesses to *Memo_Table_T* should be supported. This corroborates the results shown by Figures 11 and 12 since the number of simultaneous accesses to the *Memo_Table_T* should be related to the number of simultaneous accesses to operands in the register file. Although no combined figures considering simultaneous acesses to *Memo_Table_T* triggered by DS1 and DS2, one could assume that six (6) read ports would be a maximum number to be taken into account.

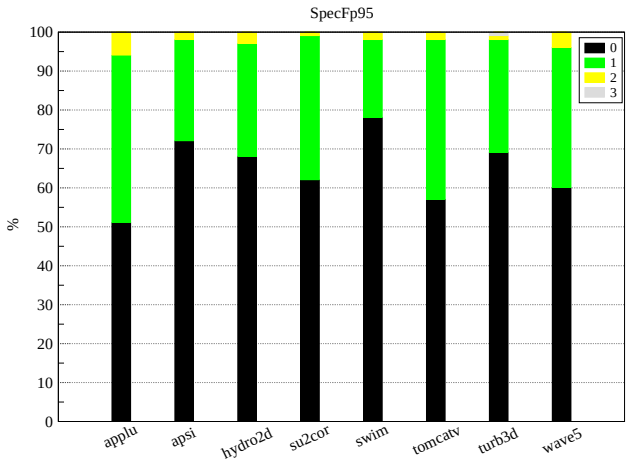

Fig. 14. Number of matching traces per clock cycle.

VI. Conclusions and Future Works

Dynamic Trace Memoization (DTM) extends the concept of instruction reuse to larger grained units by employing memoization tables to skip the execution of dynamic sequences of redundant instructions. This work presented an evaluation of critical architectural issues concerning the feasability of DTM. The cost-effectiveness of trace level reuse was investigated by showing that a balance between single instruction and trace memoization tables produces higher speedups and does not infer in a higher number of read ports in the trace memo table. It was also shown that, on the contrary to what some hypothetical situations might indicate, DTM requires no extra read ports when considering both SPECInt95 and SPECFp95 benchmark suites. In contrast to the 4% to 21% (8.4% HM) speedups obtained upon running SPECInt95, speedups from 4% to 12% (7% HM) were observed upon running the SPECFp95 benchmark suite considering a DTM mechanism having no dedicated floating-point apparatus. Such figures are, to the best of our knowledge, better than any other found in related works.

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 II). 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 [13] 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] A. Sodani, G. Sohi, *Dynamic Instruction Reuse*, Proc. of the 24th International Symposium on Computer Architecture, 1997, pp. 194–205.

[4] 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.

[5] 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.

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

[7] A. Costa, F. França, E. Chaves, *Evaluating DTM in a Superscalar Processor Architecture*, ES-498/99, COPPE/UFRJ. Rio de Janeiro,Brazil, July 2000.

[8] A. Costa, F. França, E. Chaves, *The Dynamic Trace Memoization Reuse Technique*, Proc.of The International Conference on Parallel Architectures and Compilation Techniques - PACT 2000, Philadelphia, PA, USA, October 2000, to be published.

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

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

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

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

[13] 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.

[14] 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.

[15] 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.