# Trace-Level Reuse

Antonio González[†], Jordi Tubella[†] and Carlos Molina[‡]

[†]Dpt. d'Arquitectura de Computadors
U. Politècnica de Catalunya, Barcelona, Spain
{antonio,jordit}@ac.upc.es

[‡]Dpt. d'Enginyeria Informàtica i Matemàtiques
U. Rovira i Virgili, Tarragona, Spain
cmolina@etse.urv.es

## Abstract

*Trace-level reuse is based on the observation that some traces (dynamic sequences of instructions) are frequently repeated during the execution of a program, and in many cases, the instructions that make up such traces have the same source operand values. The execution of such traces will obviously produce the same outcome and thus, their execution can be skipped if the processor records the outcome of previous executions. This paper presents an analysis of the performance potential of trace-level reuse and discusses a preliminary realistic implementation. Like instruction-level reuse, trace-level reuse can improve performance by decreasing resource contention and the latency of some instructions. However, we show that trace-level reuse is more effective than instruction-level reuse because the former can avoid fetching the instructions of reused traces. This has two important benefits: it reduces the fetch bandwidth requirements, and it increases the effective instruction window size since these instructions do not occupy window entries. Moreover, trace-level reuse can compute all at once the result of a chain of dependent instructions, which may allow the processor to avoid the serialization caused by data dependences and thus, to potentially exceed the dataflow limit.*

## 1. Introduction

Data dependences[1] are one of the most important hurdles that limit the performance of current microprocessors. The amount of instruction-level parallelism (ILP) that processors may exploit is significantly limited by the serialization caused by data dependences. This limitation is more severe for integer codes, in which data dependences are more abundant. Some studies on the ILP limits of integer applications have revealed that some of them cannot achieve more than a few tens of instructions per cycle (IPC) in an ideal processor with the sole limitation of data dependences [16]. This suggests that techniques to avoid the serialization caused by data dependences are important to boost ILP, and they will be crucial for future wide-issue microprocessors.

Two techniques have been proposed so far to avoid the serial execution of data dependent instructions: *data value speculation* and *data value reuse*. This paper focuses on the latter technique. Data value reuse is a technique that exploits the fact that many dynamic instructions or dynamic sequences of instructions (traces) are repeatedly executed, and most of these repetitions have the same inputs, and thus generate the same results. Data value reuse exploits this fact by buffering previous inputs and their corresponding outputs. When an instruction/trace is encountered again and its current inputs are found in that buffer, its execution can be avoided by getting the outputs from the buffer. This reduces the functional units utilization and, more importantly, reduces the time to compute the results, and thus, shortens the lengths of critical paths of the execution.

Techniques that try to reuse single instructions will be referred to as *instruction-level reuse*, whereas those techniques that handle dynamic sequences of instructions will be denoted by *trace-level reuse*. Data value reuse can be exploited through software or hardware mechanisms. In this work, we explore hardware techniques for trace-level reuse. Exploiting reuse at trace-level implies that a single reuse operation can skip the execution of a potentially large number of instructions. More importantly, these instructions do not need to be fetched and thus they do not consume fetch bandwidth. Finally, since these instructions are not placed in the reorder buffer[2], they do not occupy any slot of the instruction window and thus, the effective instruction window size is increased as a side effect. Particularly interesting is the fact that this technique may compute all at once the results of a chain of dependent instructions (e.g. in a single cycle), which allows the processor to exceed the dataflow limit that is inherent in the program.

In this paper, we first propose and approach to implement trace-level reuse on a superscalar processor. Then, we analyze the performance potential of such technique under different scenarios. We also compare the benefits that may be achieved by trace-level reuse with respect to instruction-level reuse. We show that trace-level reuse is more effective than instruction-level due to the reduction in fetch bandwidth and instruction window requirements outlined above. Moreover, trace-level reuse has lower overhead since a single reuse operation can avoid the execution of a long sequence of instructions.

The rest of this paper is organized as follows. Section 2 reviews the concept of data value reuse and the most relevant work. Section 3 describes a trace-level reuse mechanism. The performance of trace-level reuse and its comparison versus instruction-level reuse are analyzed in section 4. Finally, section 5 summarizes the main conclusions of this work.

1. In this paper, data dependences refer to true dependences (output and anti-dependences are not included).

2. As discussed below, some instructions are placed in the reorder buffer in order to provide precise exceptions, but in general they are much less than the number of instructions in the trace.

## 2. Related Work

The sources of instruction repetition are investigated in [13] and a study of the differences between value prediction and value reuse is presented in [14].

Data value reuse can be implemented by software or hardware. Software implementation is usually known as *memoization* or *tabulation* [2] [11]. Memoization is a code transformation technique that takes advantage of the redundant nature of computation by trading execution time for increased memory storage. The results of frequently executed sections of code (e.g. function calls, groups of statements with limited side effects) are stored in a table. Later invocations of these sections of code are preceded by a table lookup, and in case of hit, the execution of these sections of code is avoided.

A hardware implementation of data value reuse was proposed by Harbison for the Tree Machine [5]. The Tree Machine has a stack-oriented ISA and the main novelty of its architecture was that the hardware assumed a number of compiler's traditional optimizations, like common subexpression elimination and invariant removal. This is achieved by means of a *value cache*, which stores the results of dynamic sequences of code (called *phrases*). For each phrase, the value cache keeps its result as well as an identifier of its input variables. For the sake of simplicity, input variables are represented by a bit vector (called *dependence set*), such that multiple variables share the same codification. Every time that the value of a variable changes, all the value cache entries that may have this variable as an entry are invalidated.

Another hardware implementation of data value reuse is the *result cache* proposed by Richardson [10] [11]. The objective was to speed-up some long latency operations, like multiplications, divisions and square roots, by caching the results of recently executed operations. The result cache is indexed by hashing the source operand values, and for each pair of operands it contains the operation code and the corresponding result.

Result caching is further investigated by Oberman and Flynn [9]. They evaluate *division caches*, *square root caches* and *reciprocal caches*, which are similar to Richardson's result cache, but for just one type of operation: division, square root and reciprocal respectively. They also investigate a shared cache for reciprocals and square roots.

Sodani and Sohi propose the *reuse buffer* [12], which is a hardware implementation of data value reuse (or dynamic instruction reuse, as it is called in that paper). The reuse buffer is indexed by the instruction address. They propose three different reuse schemes. In the first scheme, for each instruction in the reuse buffer, it holds the source operand values and the result of the last execution of this instruction. In the second scheme, instead of the source operand values, the buffer holds the source operand names (architectural register identifiers). In the third scheme, in addition to the information of the second scheme, the buffer stores the identifiers of the producer instructions of the source operands. In this scheme, dependent instructions that are fetched simultaneously can be reused by chaining their individual reuses. However, the reuse of each individual instruction is still a sequential process since it must wait until the reuse of all previous instructions has been checked.

Jourdan *et al.* propose a renaming scheme that exploits the phenomenon of instruction-level reuse in order to reduce the register pressure. The basic idea is that several dynamic instructions that produce the same result share the same physical register [7].

Another application of data value reuse has been presented in [17]. Weinberg and Nagel describe a technique that reuses high-level language pointer-expressions with the aid of compiler inserted hints. Basically, once the input operand set of an expression matches a previously executed instance of the same expression, the result is obtained from a table instead of recomputing it.

Molina, González and Tubella presented a reused scheme referred to as *Redundant Computation Buffer* [8]. The underlying concept is the removal of redundant computations, and in particular, the run-time elimination of quasi-common subexpressions and quasi-invariants.

Finally, Huang and Lilja have recently proposed a scheme to reuse basic blocks [6]. Basic block reuse is a particular case of trace-level reuse in which traces are limited to basic blocks. Trace-level reuse is more general and can exploit reuse in larger sequences of instructions, such as subroutines, loops, etc.

## 3. Trace-Level Reuse

This section describes an approach to integrating a trace-level reuse scheme in a superscalar processor.

A trace refers to any dynamic sequence of instructions. The objective of trace-level reuse is to avoid the individual execution of the instructions in a trace. All changes in the processor state that would be produced by these instructions are done by applying again the changes that were produced in a past execution of the same trace, provided that both executions have the same inputs.

Reusing traces requires the processor to include some type of memory to store previous traces, an approach to decide which traces are worthwhile to be stored, a mechanism to identify when the forthcoming trace can be reused, and a final process to update the processor state if the trace is reusable. These issues are addressed below in more detail.

### 3.1. The Reuse Trace Memory

The reuse trace memory (RTM) is a memory that stores previous traces that are candidate to be reused. From the point of view of reuse, a trace is identified by its input and its output (see figure 1). The input of a trace is defined by: (i) the starting address, i.e. initial program counter (PC), and (ii) the set of register identifiers and memory locations that are live, and their contents before the trace is executed. A register/memory location is live if it is read before being written.

The output of a trace consists of: (i) the set of registers and memory locations that the trace writes and their contents after the trace is executed, and (ii) the address of the next instruction to be executed after the trace.

The RTM can be indexed by different schemes. For

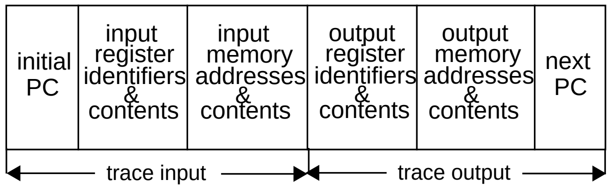
initial PC
input register identifiers & contents
input memory addresses & contents
output register identifiers & contents
output memory addresses & contents
next PC
trace input
trace output

**Figure 1: A RTM entry.**

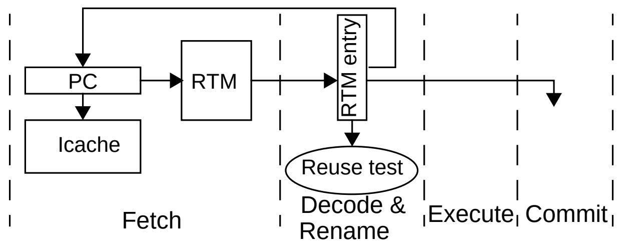

**Figure 2: Reusing a trace.**

instance, it can be indexed by PC (as considered in this paper), or by a hashing of the PC and the contents of a given register, etc.

### 3.2. Dynamic Trace Collection

The processor dynamically decides which traces of the dynamic stream are candidates to be reused. Different heuristics can be used to decide the starting and ending points of a trace. We will show later that a convenient criterion could be to start a new trace when a reusable instruction is encountered and to terminate the trace just before the first non reusable instruction is found. Another possibility that we also evaluate in this paper is to consider fixed-length traces which can be dynamically expanded once they are reused.

Note that traces may have a variable number of instructions. In fact, the instructions that make up a trace are not stored in the RTM. Obviously, there may be implementation parameters that limit the size of a trace such as the number of input and output values that can be stored in each RTM entry, but the number of instructions in a trace is not by itself a limitation.

### 3.3. Reuse Test and Processor State Update

At some points of the execution (e.g. at the fetch of the initial trace instruction, or whenever an input trace operand becomes ready) the processor checks whether the current trace can be reused. If this is the case, then the processor uses the information of the trace obtained from the RTM to update its state in the following way (see figure 2): (i) the PC is updated with the *next PC* field so that the fetch unit proceeds with the instructions that follow the trace. Instructions that belong to the trace do not need to be fetched; (ii) the output registers and output memory locations are updated with the values obtained from the RTM entry.

There are basically two approaches to identify whether a trace is reusable. One possibility is to read the current values of all input registers and memory locations and compare them with the values in any RTM entry associated to the current PC. Another possibility is to add to each RTM entry a valid bit. When a trace is stored its valid bit is set. For every register/memory write, all the RTM entries with a matching register/memory location in its input list are invalidated. The latter approach requires a much simpler reuse test (just checking the valid bit).

The final reuse process that updates the processor state can be implemented by inserting in the instruction window instructions that write the corresponding values in the trace output (registers and memory) locations. In this way, precise exceptions could be guaranteed in an out-of-order processor following the conventional mechanism.

## 4. The Performance Potential of Reuse at Instruction and Trace Levels

In this work, we are interested in studying the data value reuse phenomenon, understanding the differences between trace-level reuse and instruction-level reuse and investigating the performance potential of these techniques.

We focus on scenarios with a limited instruction window but infinite number of functional units. In this way, we do not consider the benefit of reducing functional unit contention, which due to the continuous increase in transistors per chip will have a low impact in future high-performance processors. Moreover, when the number of functional units is a bottleneck, increasing the number of functional units is more cost-effective than implementing a reuse scheme. We also consider the case of an infinite instruction window as an indication of the limits of the potential of these techniques.

For the infinite window scenario, the execution time is only limited by data dependences among instructions, both through register and memory. For the limited window scenario, the execution order inside each sequence of W instructions, W being the instruction window size, is only limited by data dependences, whereas any pair of instructions at a distance greater than W must be sequentially executed.

We compute the IPC for each different scenario as an extension of the approach proposed in [1]. The IPC for an infinite window machine is computed by analyzing the dynamic instruction stream. For each instruction, its completion time is determined as the maximum of the completion time of the producers of all its inputs plus its latency. The inputs of an instruction may be register or memory operands. Therefore, for each logical register and each memory location, the completion time of the latest instruction that has updated such storage location so far is kept in a table. The latency of the instructions has been borrowed from the latency of the Alpha 21164 instructions [3]. Once all the dynamic instruction stream has been processed, the IPC is computed as the quotient between the number of dynamic instructions and the maximum completion time of any instruction.

The process of computing the IPC for the limited instruction window scenario is an extension of the unlimited window approach. The extension consists of computing the graduation time of each instruction as the maximum completion time of any previous instruction, including itself. Then, the completion time of a given instruction is computed as the maximum among the completion time of all the producers of its inputs and the graduation time of the instruction W locations above in the trace, plus the latency of the instruction. Note that only the graduation time of the latest W instructions must be tracked.

For the performance analysis of data value reuse, we first consider a reuse engine with infinite tables to keep history of previous instructions/traces and we analyze the effect of different reuse latencies. The reuse latency corresponds to the time that a reuse operation takes. It usually involves a table lookup and some comparisons. In the last part of this section, we measure the amount of trace-level reusability when finite reuse tables are considered. In this case, different reuse trace memory sizes and dynamic trace collection heuristics are considered. The reuse test is based on an associative search of the traces that start at the same PC.

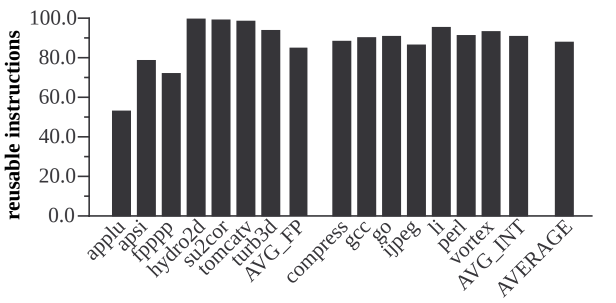

**Figure 3: Instruction-level reusability for a perfect engine.**

## 4.1. Benchmarks

The benchmark programs are a subset of the SPEC95 benchmark suite, composed of both integer and FP codes: *compress, gcc, go, ijpeg, li, perl* and *vortex* from the integer suite, and *applu, apsi, fpppp, hydro2d, su2cor, tomcatv* and *turb3d* from the FP suite.

The programs have been compiled with the DEC C and Fortran compilers with full optimization (“`-non_shared -O5 -tune ev5 -migrate -ifo`” for C codes and “`-non_shared -O5 -tune ev5`” for Fortran codes). Each program is run using the reference input for 50 millions of instructions after skipping the first 25 millions. The study of the maximum degree of reusability requires to store a huge amount of data, which prevents from analyzing the whole program execution. In this way, the results give a flavor of the overall behavior of the SPEC95 suite.

The compiled programs have been instrumented with the Atom tool [15] and their dynamic trace has been processed in order to obtain the IPC for each configuration. Results are shown for individual programs, and in some cases we show the average for integer programs, FP programs or the whole set of benchmarks. Average speed-ups have been computed through harmonic means and average percentages have been determined through arithmetic means.

## 4.2. Limits of Instruction-Level Reusability

This section is focused on measuring the potential of data value reuse at instruction-level. Hence, we consider here the maximum instruction-level reuse that can be exploited.

For each static instruction, all the different input values of its previous executed instances are stored in a table. For a given dynamic instruction, if its current inputs are the same as in a previous execution, the instruction is reusable. The percentage of reusable instructions will be referred to as the *instruction-level reusability* of a program. Note that the reusability of a program takes into account any kind of instructions, including memory accesses.

We can observe in figure 3 that instruction-level reusability is very high. For most programs it is higher than 90% of all dynamic instructions and on average it is 88%. The reusability ranges from 53% to 99%, *applu* and *hydro2d* being the programs with the lowest and highest reusability respectively. We can also observe that there are not high differences between integer and FP codes (91% and 85% of instruction-level reusability respectively). We can conclude that instruction-level reuse is abundant in all types of programs.

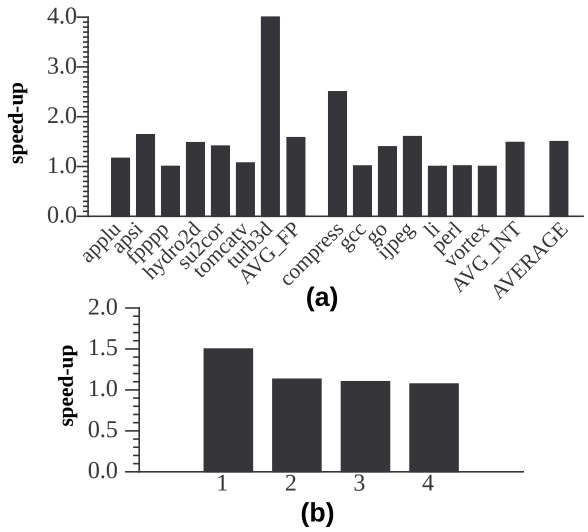

**Figure 4: Speed-up of instruction-level reuse for an infinite instruction window and: (a) a 1-cycle reuse latency, (b) a reuse latency varying from 1 to 4**

## 4.3. Performance Improvement of Instruction-Level Reuse

The ultimate figure in which we are interested is the effect of instruction-level reuse on execution time. For this scenario, the IPC is computed by extending the mechanism described for an unlimited or limited window configuration respectively. The completion time of a non-reusable instruction is computed in the same way as in the base machine, whereas the completion time of a reusable instruction is computed as the maximum of the completion time of all the producers of its inputs (an instruction cannot be reused until all its inputs are available) plus a *reuse latency.* In any case, if the completion time of a reused instruction is higher than the completion time of the normal execution of that instruction, the latter will be chosen. This is equivalent to assuming that an oracle determines the best approach for each instruction.

Figure 4.a shows the speed-up provided by instruction-level reuse when the reuse latency is assumed to be 1 cycle. Note that the speed-ups are very dependent on the particular benchmark. On average, it is around 1.50, and it is slightly higher for FP than for integer programs. However, there are some programs that can significantly benefit from instruction-level reuse, such as *turb3d* and *compress*, which show a speed-up of 4.00 and 2.50 respectively. On the other hand there are also programs that hardly benefit from instruction-level reuse, such as *fpppp* and *gcc*. In general, this performance result may look low if one takes into account the very high percentage of reusable instructions (figure 3).

Figure 4.b shows the effect of the reuse latency on performance, for a latency varying from 1 to 4 cycles per reuse (only averages are shown). Note that the benefits of instruction-level reuse significantly decrease when more than a 1-cycle latency is assumed. This indicates that the instructions that are in the critical path are usually low-latency instructions, and thus, the latency reduction achieved by instruction reuse is effective only if the reuse latency is very low. For a configuration with a limited number of functional units the benefits will be a bit higher due to the reduction in

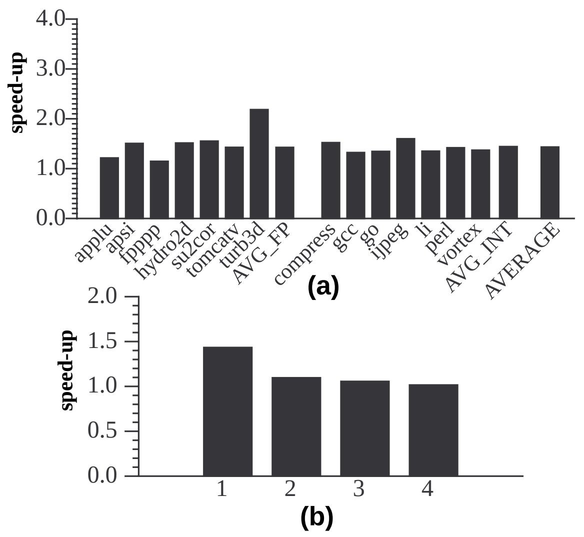

**Figure 5: Speed-up of instruction-level reuse for a 256-entry instruction window and: (a) a 1-cycle reuse latency, (b) a reuse latency varying from 1 to 4 cycles.**

functional unit contention. However, as pointed out above, adding functional units is a more cost-effective approach to reduce contention than including a reuse scheme since the latter solution is significantly more complex.

Instruction-level reuse in the case of a limited instruction window (256 entries) behaves almost in the same way as in the case of an unlimited instruction window. This is shown in figure 5.a that shows the speed-up for a 1-cycle reuse latency. On average the speed-up is 1.43, with minor difference between integer and FP codes (1.44 and 1.42 respectively). Differences among individual programs are smaller than those observed for an infinite window. The benefits for those programs that had the highest speed-ups for an unlimited instruction window (*turb3d* and *compress*) are now reduced. Finally, in figure 5.b it is shown that the benefits of instruction-level reuse when the reuse latency is greater than 1 cycle is also significantly reduced, like in the infinite window configuration (see figure 4.b).

To summarize, the benefits of instruction-level reuse are moderate for a 1-cycle reuse latency and very low for higher latencies, in spite of the fact that the percentage of reusable instructions is very high. The reason for this is that instruction reuse cannot be exploited until the source operands are ready and thus, the reuse of a chain of dependent instructions is still a sequential process.

## 4.4. Limits of Trace-Level Reusability

Reuse of traces is an attractive technique since a single reuse operation may skip the execution of a potentially long sequence of dynamic instructions, even if they are dependent among them. To evaluate the performance limits of this technique we should compute the maximum reuse that can be attained for any possible partition of the dynamic instruction stream into traces. Since there is not any constraint about the contents of each trace, the different ways to partition a dynamic instruction stream into traces are practically unlimited, which prevents an exhaustive exploration of all of them.

Given that each reuse operation has an associated latency (e.g. table lookup), the most effective schemes will be those that reuse maximum length traces. That is, given a dynamic instruction stream that corresponds to the execution of a program, we are interested in identifying a set of reusable traces such that: a) the total number of instructions included in those traces is maximum and b) the number of traces is minimum. In other words, if a trace is reusable, it is more effective to reuse the whole trace in a single reuse operation than to reuse parts of it separately. However, finding maximum length reusable traces would be still a complex problem if all the possible partitions of a program into traces should be explored.

We can however prove that if we consider just those traces that are formed by all maximum-length dynamic sequences of reusable instructions, we have an upper-bound of the reusability that can be exploited by maximum-length traces (condition (a) above) and a lower bound of the number of traces required to exploit it (condition (b) above). This is supported by the theorems below. The performance provided by assuming that such traces are reusable will provide an upper-bound of the performance limits of trace reuse.

**Theorem 1.** Let $T$ be a trace composed of the sequence of dynamic instructions $<i_1, i_2, ..., i_n>$. If $T$ is reusable, then $i_k$ is reusable for every $k \in [1,n]$.

**Proof.** Refer to the enclosed appendix in page 8**.**

**Theorem 2.** Let $T$ be a trace composed of the sequence of dynamic instructions $<i_1, i_2, ..., i_n>$. If $i_k$ is reusable for every $k \in [1,n]$, then $T$ is not necessarily reusable.

**Proof.** Refer to the enclosed appendix in page 8.

Theorem 1 implies that the number of instructions whose execution can be avoided by any trace reuse scheme is limited by the amount of individual instructions that are reusable. Thus, we can compute an upper-bound of the benefits of trace-level reuse by assuming that the amount of trace-level reusability is equal to the amount of instruction-level reusability, and the overhead of trace-level reuse is given by grouping reusable instructions into the minimum number of traces (i.e. assuming maximum-length traces). Theorem 2 states that this approach results in an upper-bound that may not be reached.

## 4.5. Performance Improvement of Trace-Level Reuse

The process to compute the IPC for this scenario is the following. The completion time of every instruction that do not belong to a reusable trace is computed in the same way as in the base machine. For a reusable trace, the completion time of all instructions that produce an output is computed as the maximum of the completion time of all the producers of its inputs plus the *reuse latency.* Moreover, two ways to consider the reuse latency have been analyzed. In one of them, the reuse latency is assumed to be a constant time per reuse operation. In the other, the reuse latency is assumed to be proportional to the number of inputs plus the number of outputs of the trace. Note that the former is more appropriate when the reuse test just requires to check a valid bit, whereas the latter models the fact that reusing a trace requires the processor to read all its inputs and check that they are the same as in a previous execution. In any case, if the completion time of an instruction in a reusable trace is higher than the completion time of the normal execution of that instruction, the latter will be chosen.

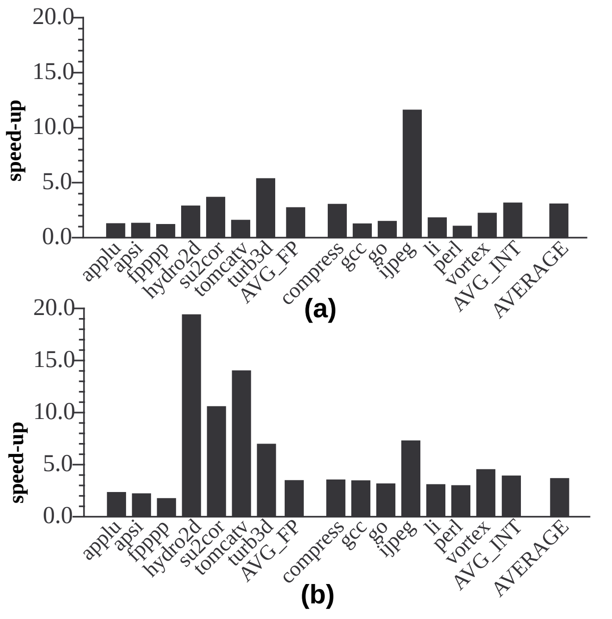

**Figure 6: Speed-up of trace-level reuse when considering a 1-cycle reuse latency for a) an infinite instruction window, b) a 256-entry instruction window.**

Performance figures of trace-level reuse are shown in figure 6. Figure 6.a corresponds to an infinite window scenario while figure 6.b is associated to a 256-entry instruction window. In both cases, a 1-cycle reuse latency has been considered. First of all, note that the average speed-up is much higher than the obtained for instruction-level reuse.

For the infinite window scenario, speed-up has increased from 1.43 to 3.03. The highest benefit is experienced by *ijpeg* (11.57). Nonetheless, there are also programs with a negligible speed-up in this scenario (*perl* with 1.01).

The difference between trace-level and instruction-level reuse is even higher for the limited window scenario. In this case, trace-level reuse may provide a very important additional advantage: it may avoid fetching instructions in reused traces and may increase the effective instruction window. If a trace is determined to be reusable (using any of the approaches described in section 3.3), the whole trace can be reused without fetching nor executing the remaining instructions of the trace. As a consequence, the speed-up of trace-level reuse for a limited instruction window is even higher than for an unlimited window (3.63 vs. 3.03), whereas for instruction-level reuse we observed the opposite trend.

It is also interesting to observe figure 7, which shows the average trace size, and correlate it with figure 6.b. Note that in general, larger traces imply higher speed-ups, which can be attributed to their higher potential to artificially increase the effective instruction window size. Integer programs have a quite uniform trace size, ranging from 14.5 to 36.7 instructions, and they also exhibit a quite homogeneous speed-up. On the other hand, some FP programs have very short traces, such as *applu*, *apsi* and *fpppp*, and they exhibit very low speed-up, whereas the others have large traces (up to 203 instructions for *hydro2d*) and they also have higher speed-ups.

It is also remarkable the fact that trace-level reuse, unlike instruction-level reuse, provides significant speed-

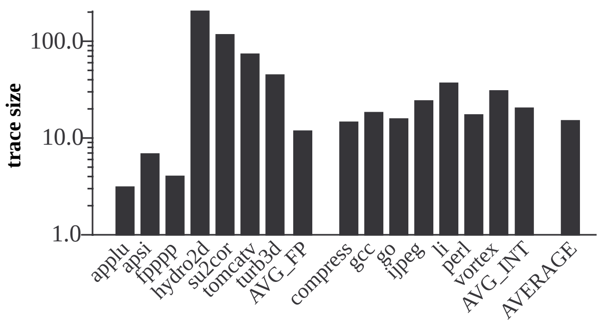

**Figure 7: Average trace size.**

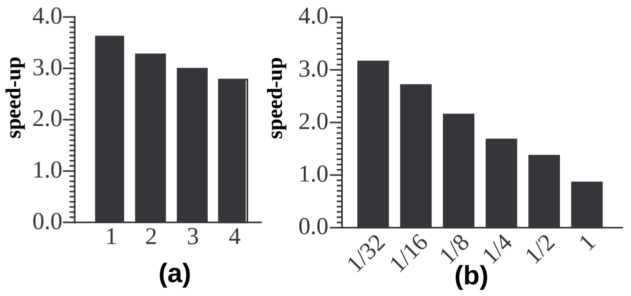

**Figure 8: Speed-up of trace-level reuse for a 256-entry instruction window and a reuse latency that a) varies from 1 to 4 cycles, b) is proportional to the number of inputs plus outputs of the trace.**

ups even if the reuse latency is higher than 1. This is shown in figure 8.a, where it can be observed that the average speed-up for a reuse latency ranging from 1 to 4 cycles is not much degraded.

Note that a trace is reused provided that all input values are the same as in a previous execution. Therefore, a trace reuse operation may imply a check of as many values as inputs the trace has. Moreover, as a consequence of a trace reuse, all the output values of the trace must be updated. Hence, it may be more realistic to assume that the reuse latency is proportional to the number of input and output values. That is, it is equal to a constant *K* multiplied by the number of input/output values. *K* is the inverse of the read/write bandwidth of the reuse engine; for instance, *K*=1/16 implies that the reuse engine can read or write 16 values per cycle. Under this scenario, the speed-up of trace-level reuse is shown in figure 8.b, where the X axis represents different values of *K*. It can be observed that the speed-up of value reuse is still high, although it is significantly affected by the reuse latency. Note that it is reasonable to assume that future microprocessors may have the capability to perform around 16 reads+writes per cycle, including register and memory values. In fact, current microprocessors such as the Alpha 21264 [4] can perform 14 reads+writes per cycle (8 register reads, 4 register writes and 2 memory references). Thus, looking at the bar corresponding to *K*=1/16 in figure 8.b, we can conclude that a speed-up around 2.7 is reasonable to be expected from trace-level reuse. This also suggests that even the slowest approach to checking reusability that is based on comparing all inputs (see section 3.3) can significantly improve performance.

On average, we have measured that the number of input values per trace is 6.5 (2.7 register values and 3.8 memory values), and the number of output values is 5.0 (3.3 register values and 1.7 memory values). Since the average number of instructions per trace is 15.0, this means that each reused instruction requires 0.43 reads and 0.33 writes, which is significantly lower than the number of reads and writes required by the execution of an instruction. We can thus conclude that trace-level reuse also provides a significant reduction in the data bandwidth requirements, and thereby it can reduce the pressure on the memory and register file ports.

## 4.6. Trace-Level Reusability with Finite Tables

In the previous section we have demonstrated the high potential of trace-level reuse. In the final part of this work we evaluate a realistic approach that implements this concept. The objective is to measure the percentage of reusability and the average trace size that this technique can provide when a finite reuse memory and a particular heuristic for trace collection are considered.

We have evaluated different capacities for the Reuse Trace Memory (RTM):

- 512 entries: A 4-way set-associative memory (5-bit index) with 4 entries per initial PC. This means that up to 4 different traces starting at the same PC can be stored.
- 4K entries: A 4-way set-associative memory (7-bit index) with 8 entries per initial PC.
- 32K entries: A 8-way set-associative memory (8-bit index) with 16 entries per initial PC.
- 256K entries: A 8-way set-associative memory (11-bit index) with 16 entries per initial PC.

In all cases, the memory is indexed by the least-significant bits of the PC register. Replacement policy is LRU, that is, the older trace with the same PC that has been reused is the one that is being replaced when a new trace is collected. For each trace, the number of inputs and outputs have been limited to 8 registers and 4 memory values.

Three different heuristics for dynamic trace collection have been considered:

- ILR NE: A trace consists of a sequence of dynamic instructions that are reusable at instruction-level. In this case, a different reuse memory used for testing instruction-level reusability is also needed. This memory has as many entries as the RTM.
- ILR EXP: The same as before with the difference that traces can be dynamically expanded when two consecutive traces are reused or instructions following a reused trace become reusable.
- I(*n*) EXP: A trace is formed by a fixed number of *n* instructions. When a trace is reused, it is expanded with *n* new instructions.

The reuse test is performed for every fetch operation. When a trace beginning at a given PC contains the same values in its input locations as the current ones, the trace is reusable. Figure 9 shows the percentage of reusable instructions and the average trace size for each scheme. First, we can observe that dynamic trace expansion is an important issue to increase the granularity of reusable traces while the total amount of reusability remains almost constant (see heuristics ILR NE and ILR EXP). Note also that heuristic I(n) outperforms ILR, that is, a policy to collect traces should consider any kind of instructions rather than those reusable at instruction level.

Another important figure is the relation between the

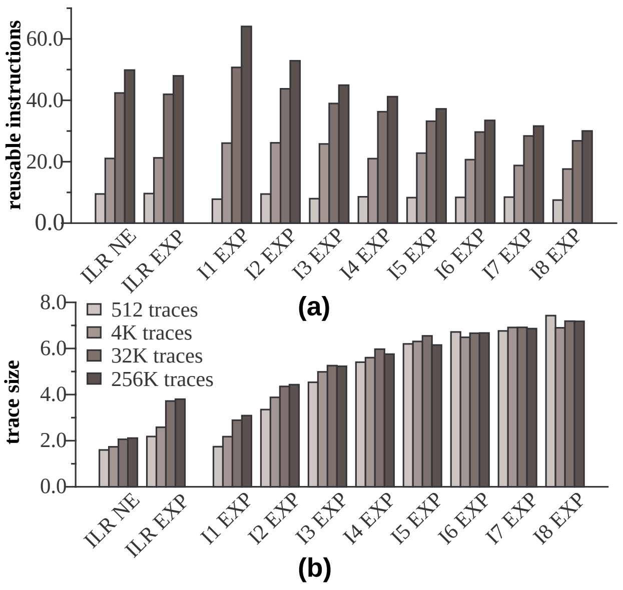

**Figure 9: Trace-level reusability with a realistic implementation. a) Percentage of reusable instructions. b) Average trace size.**

RTM size and the achieved reusability. For instance, a 4K-entry RTM can reuse 25% of the dynamic instructions with an average trace size of 6 instructions. The percentage of reused instructions significantly grows with the RTM capacity. Finally, note the trade-off between percentage of reused instructions and trace size. Increasing the trace size reduces the number of reused instructions. However, to achieve a given degree of reuse, the reuse overhead is reduced when the trace size increases.

## 5. Conclusions

We have presented a trace-level reuse mechanism and analyzed its performance. We have shown that trace-level reuse can reuse a lower percentage of instructions than instruction-level reuse. However, the former is more effective because of several reasons: a) it reduces the fetch bandwidth requirement by avoiding fetching instructions of reused traces; b) it increases the effective instruction window size by avoiding storing instructions of reused traces in the instruction window; c) it has a lower overhead since it requires a lower number of operations per reused instruction.

For a 256-entry instruction window and infinite history tables, trace-level reuse provides a speed-up of 3.6 in average, which ranges from 1.7 to 19.4 for individual programs, when the reuse latency is 1 cycle. Similar results are obtained when the reuse latency is considered proportional to the number of inputs and outputs of a trace.

Finally, we have evaluated the impact of a limited-capacity history table. For instance, for a 4K-entry Reuse Trace Memory we have observed that the percentage of reusability is around 25% of all dynamic instructions while the average trace size is around 6 instructions. For a 256K-entry Reuse Trace Memory, around 60% of instructions can be reused.

## Acknowledgements

This work has been supported by grants CICYT TIC 511/98 and ESPRIT 24942. The research described in this paper has been developed using the resources of the European Center for Parallelism of Barcelona (CEPBA).

## References

[1] T.M. Austin and G.S. Sohi, “Dynamic Dependence Analysis of Ordinary Programs”, in *Proc. of Int. Symp. on Computer Architecture*, pp. 342-351, 1992

[2] H. Abelson and G.J. Sussman, *Structure and Interpretation of Computer Programs*, McGraw Hill, New York, 1985

[3] Digital Equipment Corporation, *Alpha 21164 Microprocessor. Hardware Reference Manual*. 1995

[4] L. Gwennap, “Digital 21264 Sets New Standard”, *Microprocessor Report*, vol. 10, no. 14, Oct. 1996.

[5] S.H. Harbison, “An Architectural Alternative to Optimizing Compilers”, in *Proc. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* 1982

[6] J. Huang and D. Lilja, “Exploiting Basic Block Value Locality with Block Reuse”, in *Proc. of 5th. Int. Symp. on High-Performance Computer Architecture,* 1999

[7] S. Jourdan, R. Ronen, M. Kekerman, B. Shormar and A. Yoaz, “A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification” in *Proc. of 31st. Ann. Int. Symp. on Microarchitecture,* 1998

[8] C. Molina, A. González and J. Tubella, “Dynamic Removal of Redundant Computations” in *Proc. of the ACM Int. Conf. on Supercomputing,* Rhodes (Greece), June 1999

[9] S.F. Oberman and M.J. Flynn, “On Division and Reciprocal Caches”, Technical Report CSL-TR-95-666, Stanford University, 1995

[10] S. E. Richardson, “Exploiting Trivial and Redundant Computations”, in *Proc. of Symp. on Computer Arithmetic*, pp. 220-227, 1993

[11] S. E. Richardson, “Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation”, Technical Report SMLI TR-92-1, Sun Microsystems Laboratories, 1992

[12] A. Sodani and G.S. Sohi, “Dynamic Instruction Reuse”, in *Proc. of Int. Symp. on Computer Architecture,* 1997

[13] A. Sodani and G.S. Sohi, “An Empirical Analysis of Instruction Repetition” in *Proc. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems”*, 1998

[14] A. Sodani and G.S. Sohi, “Understanding the Differences Between Value Prediction and Instruction Reuse”, in *Proc. of 31st. Ann. Int. Symp. on Microarchitecture,* 1998

[15] A. Srivastava and A. Eustace, “ATOM: A system for building customized program analysis tools” , in *Proc of the 1994 Conf. on Programming Languages Design and Implementation,* 1994.

[16] D.W. Wall, “Limits of Instruction-Level Parallelism”, Technical Report WRL 93/6, Digital Western Research Laboratory, 1993.

[17] N. Weinberg and D. Nagle, “Dynamic Elimination of Pointer-Expressions”, in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques,* pp. 142-147, 1998.

## A. Appendix

In this appendix we present the proof of theorems 1 and 2. In fact, we will prove a more general formulation of theorem 1 and 2, in which, a trace is considered to be a sequence of consecutive traces of a smaller size. Theorem 1 and 2 are the particular case in which the size of every smaller trace is one instruction.

**Definitions.** An *input* of a trace $T$ is a register, condition code or memory location that is read and has not been written before in such trace. An *output* of a trace $T$ is a register, condition code or memory location that is written in such trace. Let $IL(T)$ be the sequence of *input* storage locations of trace $T$. Notice that $IL(T)$ is a sequence and not a set. The order of the sequence is given by the order in which the inputs are read. Let $OL(T)$ be the sequence of *output* storage locations of trace $T$. The order of the sequence is given by the order in which the outputs are written. Let $IV(T)$ be the sequence of *input* values of trace $T$, in the order in which they are read. Let $OV(T)$ be the sequence of *output* values of trace T, in the order in which they are written.

If $A$ and $B$ are two sequences, we will say that $A \subseteq B$ if $A$ is a subsequence of $B$. Moreover, $A \cup B$ will refer to any sequence that is composed of the elements of $A$ and $B$, no matter the order of the elements. Different dynamic instances of the same trace will be denoted by using the same symbol to refer to the trace, with a superscript that corresponds to the dynamic execution order. Notice that different instances of the same trace will always have the same input/output registers but may have different input/output memory locations.

If a trace $T$ is reusable, it must happen that $IL(T^i) = IL(T^j)$ and $IV(T^i) = IV(T^j)$ for some $j < i$. This obviously implies that $OL(T^i) = OL(T^j)$ and $OV(T^i) = OV(T^j)$. That is, if the inputs are the same and have the same value, then the outputs will also be the same and will have the same values.

**Theorem 3 (generalization of theorem 1).** Let $T$ be a trace composed of a sequence of traces $<t_1, t_2, ..., t_n>$. If $T$ is reusable, then $t_k$ is reusable for every $k \in [1,n]$.

**Proof.** If $T^i$ is reusable, then $IL(T^i) = IL(T^j)$ and $IV(T^i) = IV(T^j)$ for some $j < i$. Notice that $IL(t_1{}^i) \subseteq IL(T^i) = IL(T^j)$, which implies that $IL(t_1{}^i) = IL(t_1{}^j)$ and $IV(t_1{}^i) = IV(t_1{}^j)$. Therefore, $OL(t_1{}^i) = OL(t_1{}^j)$ and $OV(t_1{}^i) = OV(t_1{}^j)$. Thus, $t_1$ is reusable.

Notice that $IL(t_2{}^i) \subseteq IL(T^i) \cup OL(t_1{}^i) = IL(T^j) \cup OL(t_1{}^j)$. Since $IL(t_2{}^j) \subseteq IL(T^j) \cup OL(t_1{}^j)$ and $OV(t_1{}^i) = OV(t_1{}^j)$, we have that $IL(t_2{}^i) = IL(t_2{}^j)$ and $IV(t_2{}^i) = IV(t_2{}^j)$. Thus, $t_2$ is reusable, that is, $OL(t_2{}^i) = OL(t_2{}^j)$ and $OV(t_2{}^i) = OV(t_2{}^j)$.

In general, we can prove that $t_{k+1}$ is reusable provided that $t_i$ is reusable for any $i=1..k$. Notice that $IL(t_{k+1}{}^i) \subseteq IL(T^i) \cup OL(t_k{}^i) \cup OL(t_{k-1}{}^i) \cup ... \cup OL(t_1{}^i) = IL(T^j) \cup OL(t_k{}^j) \cup OL(t_{k-1}{}^j) \cup ... \cup OL(t_1{}^j)$. Thus, $IL(t_{k+1}{}^i) = IL(t_{k+1}{}^j)$ and $IV(t_{k+1}{}^i) = IV(t_{k+1}{}^j)$, which means that $t_{k+1}$ is reusable.

**Theorem 4 (generalization of theorem 2).** Let $T$ be a trace composed of the sequence of traces $<t_1, t_2, ..., t_n>$. If $t_k$ is reusable for every $k \in [1,n]$, then $T$ is not necessarily reusable.

**Proof.** If $t_1{}^i, t_2{}^i, ..., t_n{}^i$ are reusable, then each of them have their inputs equal to those of some previous execution. That is, $IV(t_1{}^i) = IV(t_1{}^{j1})$, $IV(t_2{}^i) = IV(t_2{}^{j2})$ , ..., $IV(t_n{}^i) = IV(t_n{}^{jn})$, but $j1, j2, ..., jn$ may be different. Therefore, $IV(T^i) \subseteq IV(t_1{}^i) \cup IV(t_2{}^i) \cup ... \cup IV(t_n{}^i) = IV(t_1{}^{j1}) \cup IV(t_2{}^{j2}) \cup ... \cup IV(t_n{}^{jn})$, but $IV(T^i)$ may be different from $IV(T^j)$ for every $j < i$, and thus, $T^i$ may be non-reusable.