Dynamic Removal of Redundant Computations

Carlos Molina, Antonio González and Jordi Tubella

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya,
Jordi Girona 1-3, Edifici D6,
08034 Barcelona, Spain
E-mail: {cmolina,antonio,jordit}@ac.upc.es

Abstract

A mechanism for dynamic instruction-level reuse in superscalar microprocessors is presented. The underlying concept that the mechanism exploits is the run-time removal of redundant computations, and in particular the elimination of common subexpressions and invariants. Removing redundant computation is a target of optimizing compilers but sometimes they do not succeed due to their limited knowledge of the data. Moreover, the proposed mechanism can also remove quasi-redundant computations, such as subexpressions that often produce the same result but sometimes they differ, depending on the data values, and thus they cannot be eliminated by the compiler. Experimental results for the Spec95 show that on average the mechanism can avoid the execution of about 32% of the dynamic instructions and provides an 1.10 speedup in a superscalar microprocessor. An extensive evaluation of different configurations and a comparison with previous schemes are presented, as well as the performance potential of a perfect reuse engine.

Keywords:

Data-value reuse, instruction-level reuse, instruction-level parallelism.

1. Introduction

Computations performed by programs tend to be repetitious in the sense that most of the data consumed/produced by different dynamic instructions are often the same [12] [4]. This is a consequence of the concise and structured way in which programs are written. For instance, different activations of the same routine may result in the repetition of some instructions with the same source operands if just a subset of the input parameters varies. Those techniques that exploit this phenomenon by reusing previous computed data instead of recomputing them again will be referred to as *data-value reuse* techniques. *Instruction-level reuse* is a particular implementation of data-value reuse that seeks to avoid the execution of instructions that produce the same result as some previous instructions. A given dynamic instruction can reuse the result of a previous instance of the same static instruction or an instance of any other static instruction. The former case corresponds to the removal of a *quasi-invariant*, which is defined as a computation that is repeated many times and often produces the same result. The second case corresponds to the elimination of a *quasi-common subexpression*, which is defined as a computation that often produces the same result as another piece of code. An example is given in Figure 1. If both arrays *b* and *c* happen to have many repeated elements, the computation `b[i]+c[i]` will become quasi-invariant at run time. Similarly, if *t* and *u* often have the same value, then the expressions `s/t` and `s/u` will be quasi-common subexpressions.

In this paper, we propose a mechanism that can take advantage of both types of reuse, and thus it can remove at run time quasi-invariants and eliminate quasi-common subexpressions. We show that the proposed scheme outperforms previous proposals in terms of reduction of the execution time. The paper also proposes some extensions to previous schemes and presents a detailed evaluation of them to determine the best configuration for a given cost.

Instruction-level reuse has two positive effects: it lowers the latency of some instructions and reduces the contention of the processor resources since reused instructions do not go through the issue and execution phases of the pipeline. Note that instruction-level reuse has the same objective as some compiler optimizations. However, the compiler can just remove some redundant computations due to its limited knowledge of the data. It usually cannot identify quasi-redundant computations, such as those examples in Figure 1.

The rest of this paper is organized as follows. Section 2 reviews previous related work. The proposed mechanism is described in section 3. Section 4 presents some extensions to previously proposed mechanisms. A timing analysis of the different reuse schemes is presented in section 5. Performance figures of the different schemes are discussed in section 6. An enhancement to the proposed mechanism that is based on a hybrid approach is presented and evaluated in section 7. Section 8 seeks to gain insight into the performance potential of instruction-level reuse by evaluating the speedup achieved by a perfect reuse engine. Finally, section 9 summarizes the conclusions of this work.

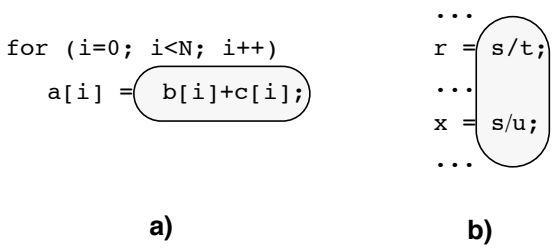

Figure 1. Examples of a) quasi-invariant; and b) quasi-common subexpression.

2. Related Work

Some techniques to exploit data value reuse have been previously proposed. Note that data value reuse can be implemented by software or hardware. Software implementations are usually known as *memoization* or *tabulation* [1] [7] [10]. Memoization takes advantage of the redundant nature of computation by trading execution time for increased memory storage. The main idea is to store the result of functions together with their input values in a table that exists as a software data structure. This technique can also be extended to statements, groups of statements, or any given region that has limited side effects. Later invocations of these sections of code are preceded by a table lookup and in case of hit, their executions are avoided.

This paper focuses on hardware implementations. Harbison proposes a *value cache* for the Tree Machine [5], which is a stack-oriented machine. The value cache stores the results of previous dynamic sequences of code (called *phrases*) together with information about the variable on which each sequence depends. New executions of the same phrases can be reused provided that none of the inputs variables have been written since the previous execution.

Richardson proposes a *result cache*, which seeks to speedup some long latency floating point operations, like multiplications and divisions [9] [10]. It consists of a buffer that is indexed by hashing the source operand values, and contains the last operation applied to such values and its result. Result caching is further investigated by Oberman and Flynn [8], which propose a specific buffer for each type of long-latency operations: *division caches, square root caches* and *reciprocal caches*. It is also evaluated by Azam *et al*. [2] as a mechanism to reduce power consumption.

Sodani and Sohi propose the *reuse buffer* [11], which is indexed by the instruction address. Three different schemes are proposed. In the first scheme, each entry of the buffer contains the source operands and the result of the last dynamic instance of the corresponding instruction. In the second scheme, each entry contains the source register identifiers and the result. In the third scheme, each entry contains pointers to the instructions that compute the source operands, in addition to the fields of scheme 2. Experimental results show that the first scheme is the one that provides the highest speedup for large reuse buffer sizes. A set-associative implementation of the reuse buffer allows it to store multiple dynamic instances of the same static instruction.

Sodani and Sohi present a study of the phenomenon of instruction repetition [12]. This study points out that many dynamic instructions in programs are repeated (between 57% and 99%, depending on the program. They also analyze the sources of such phenomenon. These authors also present a comparative study of value prediction and instruction reuse in [13].

González et al. study the performance limits of data-value reuse for two different approaches: instruction-level reuse and trace-level reuse [4].

Weinberg and Nagle proposed a mechanism to eliminate the computation of pointer expressions that consists of a history table that stores source operands and results of previously evaluated pointer expressions. The mechanism requires some compiler support to mark the instructions that are involved in the expressions [16].

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.

3. The Redundant Computation Buffer

This section presents the proposed mechanism to exploit instruction-level reuse. It is based on a buffer that stores information about which computations are likely to be redundant. This buffer is referred to as the *redundant computation buffer (RCB)*.

The performance of a reuse mechanism not only depends on the percentage of reused instructions but it also depends on the *reuse latency*. This latter term refers to the time spent since an instruction is fetched until the reuse engine produces its result.

Note that an important difference between the result cache and the reuse buffer (see section 2) is that the former requires the source operands to index the buffer, whereas the latter is indexed by the instruction address and its operands are just required once the entry is read, in order to be compared with the operands in the entry. In other words, the reuse buffer can overlap the buffer lookup with the instruction fetch, register rename and register read operations, whereas the result cache cannot. Therefore, if we assume that a table lookup takes one cycle, the reuse buffer mechanism can produce the result of a reusable instruction one cycle earlier than the result cache (see section 5 for a detailed discussion of timing considerations). On the other hand, the main drawback of the reuse buffer is that it can only reuse dynamic instances of the same static instruction. In other words, the reuse buffer seeks to exploit quasi-invariants but does not take advantage of quasi-common subexpressions (see definitions in section 1).

Moreover, the reuse buffer can reuse multiple instructions fetched simultaneously and dependent among them. Since the reuse buffer is indexed by the instruction address, the corresponding multiple entries can be read simultaneously and the reused result of an instruction can be used to check if the following dependent instructions can be reused, in a similar way as register renaming of multiple dependent instructions can take place in the same cycle. We refer to this feature as *reuse chaining*. On the other hand, reuse chaining cannot be exploited by the result cache, since the buffer lookups would be sequential and could not be performed in a single cycle. This is due to the fact that the result cache uses the source operand values to index the buffer and thus, if instruction B depends on instruction A, the buffer lookup of instruction B cannot be performed until the result of instruction A is available (i.e., until the reuse of A finishes and produces the source operand of B).

The RCB proposed in this work seeks to eliminate redundant computation resulting from both quasi-invariants and quasi-common subexpressions. Moreover, it is indexed by the instruction address and thereby it has the same reuse latency as the reuse buffer and it can also exploit reuse chaining.

Note that when two different static instructions produce the same values, the one that produces them later could reuse the computation performed by the other (i.e. the reuse is just in one direction). We will refer to the former instruction as the *consumer* and to the latter as the *producer*.

The RCB is shown in Figure 2. It consists of three tables: one seeks to reuse arithmetic instruction results and memory addresses (Atable); another tries to reuse load values (Mtable); and a third one that is used to identify quasi-common subexpressions (Vtable). The Atable is indexed by the instruction address and each entry contains the following fields:

- The opcode of the instruction.
- Two operand values (opnd1 and opnd2).

- A result value, which corresponds to either the result of the latest arithmetic instruction or the address of the latest memory operation that was mapped onto that entry.
- A pointer to the entry that stores the results of the instruction that produces values that may be reused by the instruction mapped onto this entry; in other words, a pointer to the producer according to the above introduced terminology.

Note that entries do not include a tag. An instruction can reuse the result of a given entry provided that the actual source operands and the opcode match those in the entry, no matter to which static instruction the entry corresponds. Thus, unlike the reuse buffer [11], the Atable does not include the PC tag but it includes the opcode, which is shorter than the tag. This not only reduces the table size but it also allows to take advantage of interferences among instructions with the same opcode. Interfering instructions with different opcodes will not be able to exploit reuse and thus, they will have a negative effect (these interferences would also occur if PC tags were used instead of opcodes). Although we have measured that this optimization slightly improves the percentage of reused instructions (since there are more destructive than constructive interferences) it comes at no cost. In fact, it allows to reduce the cost of the Atable by avoiding to store the tags. When comparing the RCB and the reuse buffer, we will consider an enhanced reuse buffer that also includes this optimization.

The Mtable is indexed by the effective address of memory operations (obtained from the Atable) and each entry contains the latest value read from / write to that location, in addition to a tag that identifies the effective address. In fact, this table acts in a similar way to a cache memory and it is not an essential part of the mechanism. The main benefit of such a table in front of getting the same data from the memory hierarchy is its potential shorter access time (due to its small capacity) and the reduction in cache ports pressure.

The Vtable stores information about the latest results of instructions. This table provides for each result value recently produced an identifier of the producer instruction. The table is indexed by the result value and, in a first approximation, each entry would contain the opcode and the address of the instruction that produced that value. However we have observed by experimental evaluation that just storing the index of the Atable where the instruction that produces the value is mapped, obtains similar performance, with a significant saving in storage.

The RCB works as follows. The Atable is accessed twice

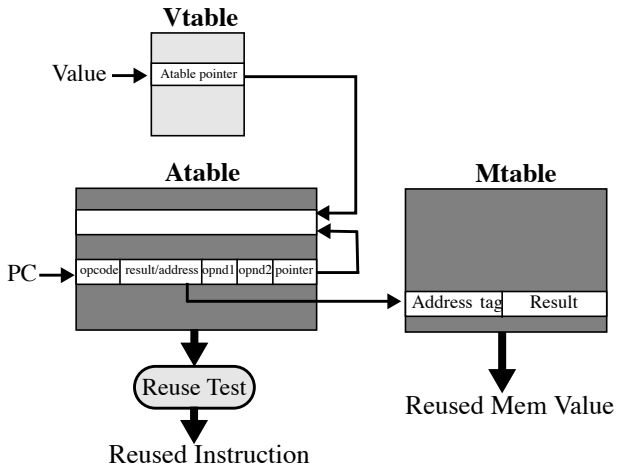

Figure 2. The redundant computation buffer (RCB).

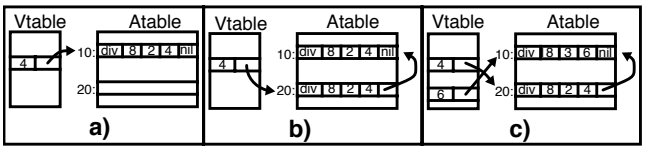

Figure 3. A working example. Contents of the tables a) after first execution of *I1*, b) after first execution of *I2*, and c) after second execution of *I1*.

while the instruction is fetched and decoded. The first probe uses the instruction address and gets the potential result if the instruction happens to produce the same result as the last execution of the same static instruction (or another static instruction that is mapped onto the same entry). It also gets the pointer to the producer instruction in case it may reuse from another static instruction. The latest operands and result of that producer are obtained by accessing the table again. Once the instruction has been decoded and the source operands have been read, the actual source operands are compared with both the latest operands of the same static instruction and the latest operands and the opcode of the producer instruction. If any of the entries match, the instruction bypasses the issue and execute stages and goes directly to the write-back stage.

Load instructions may reuse the effective address using the Atable. In addition, they may reuse their load values by means of the Mtable. This latter table is accessed by load instructions that have managed to reuse their addresses and is indexed by the reused address[1]. Store instructions can also reuse their effective addresses by means of the Atable. Loads update the Mtable when they are executed and stores do it when they are committed.

The Vtable, which is indexed by the result value, is updated by every instruction in the commit stage. Before writing to this table, every instruction first reads the corresponding entry and checks if this entry was previously updated by another instruction with the same opcode. In this case, a pointer from the consumer (current instruction) to the producer (instruction found in the Vtable) is set in the Atable[2].

3.1 A Working Example

This section illustrates how the mechanism achieves the reuse of quasi-common subexpressions. For this purpose, let us assume the following code:

```
while (cond) {
    r = s/t;
    ...
    x = s/u;}
```

Let *I1* and *I2* denote the instructions that compute the values of *r* and *x* respectively, which happen to be quasi-common subexpressions, and let us assume that their PC is 10 and 20 respectively. The first time that *I1* is executed the values of *s* and *t* are 8 and 2 respectively and thus, the result of the division is 4. The Atable entry indexed by 10 (PC of *I1*) is updated to reflect this operation and the Vtable entry indexed by the result is made to point to this Atable entry, as it is shown in Figure 3.a. When *I2* is

[1] The conventional hardware disambiguation mechanism is used to enforce memory dependences. If there is a previous uncommitted store to the same address, the load does not access the Mtable.

[2] Note that different schemes to assign confidence to the reuse among different static instruction may be devised. For instance, by means of saturating counters we could just establish a link from a producer to a consumer after a number of consecutive reuses. The study of such confidence mechanisms is beyond the scope of this work.

executed for the first time it cannot be reused since the Atable entry indexed by 20 (PC of *I2*) is empty. Its source operands are also 8 and 2. Once the instruction completes, the Atable is updated in a similar way as explained above for *I1*. Moreover, the Vtable is indexed by the result (4) and since it is found in the table, the previous instruction that produced the same result (*I1*) is obtained following the pointer. At this point, a link from *I2* to *I1* in the Atable is established by just copying the pointer found in the Vtable into the Atable entry corresponding to *I2*, as shown in Figure 3.b. This link indicates that *I2* has produced the same result as *I1* and thus *I2* is candidate to reuse results from *I1* in the future.

Next time that *I1* is executed, its source operands are 18 and 3. Since they are not the same as in the last execution, it cannot be reused. When the instruction completes, the entry 10 of the Atable is updated to reflect the new source operands and the new result, and the Vtable is also updated to indicate the latest producer of the value 6, as shown in Figure 3.c. When instruction *I2* is encountered again, its source operands are equal to the last operands of *I1* (i.e. 18 and 3). In the decode stage the corresponding Atable entry is looked up to check whether it can reuse the result of its previous execution, which is not the case. Since a link to instruction *I1* is found, it is also checked whether it can reuse the result of *I1*, as it is the case since current source operands of *I2* match the last source operands of *I1*. Then, the Atable entry corresponding to *I2* is updated with the current source and destination operands (not shown in Figure 3).

This process is repeated for following executions of *I1* and *I2*. Every execution of *I1* updates its Atable entry whereas every execution of *I2* reuses the value that it founds in the entry corresponding to *I1*. Note also that reuse can be exploited no matter how complex the quasi-common subexpression is. For complex subexpressions, reuse will occur for the instruction that computes the final result, as well as those instructions that compute intermediate values.

3.2 Reusing From Non-Latest Results

Note that the previous scheme allows the processor to reuse the result of the latest execution of instructions. However, in some cases, instructions can reuse results from previous executions of either the same or another static instruction. For instance, in the following code:

```
for (i=0; i<N; i++){
     ...
     x = a[i]+b[i];                (S1)
     y = a[i-2]+b[i-2];            (S2)
     ... }
```

the result of statement S2 is the same as statement S1 of two iterations earlier (assuming that arrays *a* and *b* are not modified in the loop).

This fact can be exploited by storing the last *N* results for each instruction, instead of the latest one. We will refer to *N* as the *history depth*. This can be implemented by means of a N-way set-associative buffer, as proposed in [11].

4. Enhanced Result Cache and Enhanced Reuse Buffer

The RCB mechanism will be compared against previous proposals for superscalar processors (the result cache and the reuse buffer), which are enhanced with some features of the RCB that can also be applied to these schemes.

The result cache consists of a buffer (Atable) that is indexed by hashing the source operand values and the opcode[1]. Each entry contains the source operands and the result. It has been enhanced with a table to store memory values, which has the same structure as the Mtable of the RCB scheme.

The Enhanced Reuse Buffer consists of two tables. One table is targeted to reuse arithmetic instructions, conditional branches and memory addresses (Atable) and the other seeks to reuse memory values (Mtable). In the original proposal [11], these two tables were merged in a single one but the authors already suggested that a split implementation could be more cost-effective. Nevertheless, they suggested a separate table for memory instructions that was indexed by the instruction address. That implementation required that store instructions associatively searched the table for a matching memory address, which is avoided by the Mtable. Moreover, like the RCB, the instruction address tag is replaced by the opcode. This allows to take advantage of constructive interference as well as it reduces the required storage, as discussed in section 3.

Note that the result cache can exploit reuse from both quasi-invariants and quasi-common subexpressions. Moreover, it can exploit reuse from non-latest results. On the other hand, the reuse buffer can just exploit quasi-invariants. However, since it is indexed by the instruction address, its reuse latency is shorter than that of the result cache. We will show in the evaluation section that the RCB takes the best of both worlds: it exploits both quasi-invariants and quasi-common subexpressions and it has the same reuse latency as the reuse buffer.

5. Timing considerations

As pointed out above, the performance of a reuse scheme is determined not only by the percentage of reused instructions but also by the reuse latency. Let us assume a dynamically scheduled processor with a microarchitecture that keeps speculative results in the reorder buffer or rename buffers and is pipelined in the stages shown in Figure 4.a (this example is based on the PowerPC 604 [15] although the conclusions are the same for other pipelines). Every instruction is fetched and then it is decoded and the physical location that holds the last definition of each source operand (if available) is identified. Then, available operands are read and the instruction dispatched to a reservation station. When all the operands are ready and a functional unit is available, the instruction is issued. Then it is executed and the result is written

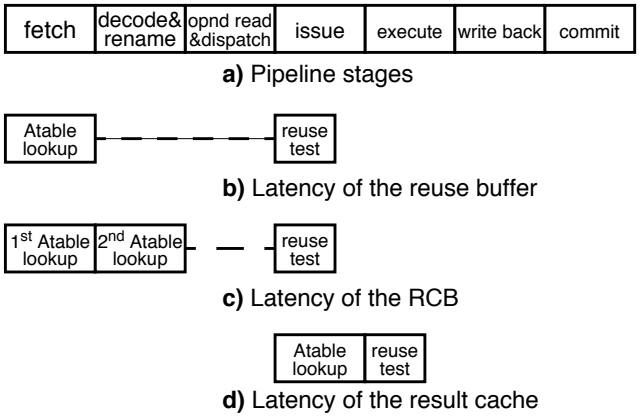

Figure 4. Reuse latency of the reuse buffer, the RCB and the result cache.

[1] Each index bit is obtained by X-oring one bit from the operand 1, one bit from the operand 2 and another from the opcode. Several X-oring schemes have been tried and we present results for the most effective one. However, the analysis of other hashing schemes may be an interesting area for further research.

back. Finally, instructions commit in order.

The reuse buffer (see Figure 4.b) accesses the Atable in parallel with the Icache lookup, since it is indexed by the PC. However, the reuse is not effective until the reuse test is performed. This test cannot be done until the source operands are read. Since the reuse test involves just equality comparisons, it is reasonable to assume that it takes less than a cycle.

The RCB (see Figure 4.c) accesses the Atable in the fetch stage by indexing it through the PC. As a result of this access, it obtains a pointer to a potential producer of the same result (a single link is provided no matter which history depth is implemented). In the next cycle, it accesses the Atable entry corresponding to this pointer. The reuse test is performed once the operands are read and thus, the total reuse latency is the same as that of the reuse buffer.

The result cache (see Figure 4.d) indexes the Atable by means of the operand values, and thus it cannot perform the access until the operands are read. This implies that the latency of this mechanism is one cycle longer than that of the reuse buffer and the RCB.

For all the reuse schemes, once the reuse test is successful, the instruction bypasses the execute stage and goes directly to the write-back stage.

6. Performance Evaluation

This section evaluates the performance of the RCB and compares it with that of the result cache and the reuse buffer.

6.1 Experimental Framework

The different reuse schemes have been evaluated for a superscalar processor using the Alpha version of the SimpleScalar toolset [3]. This is an execution-driven simulator based upon the Alpha ISA.

The following Spec95 benchmarks have been considered: *compress, go, gcc, li, m88ksim, perl* and *vortex* from the integer suite; and *applu, mgrid, swim* and *turb3d* from the FP suite. The programs have been compiled with the DEC C and F77 compilers with `-non_shared -O5` optimization flags (i.e, maximum optimization level). Each program was run with the test input set and statistics were collected for the first 125 million of instructions after skipping the initial part corresponding to initializations.

N-way set associative buffers, where N is the history depth, are considered for all the reuse mechanisms. The performance of each scheme is drawn in front of its required storage capacity, which is measured as the total number of bits required to implement it, including tags when they are used.

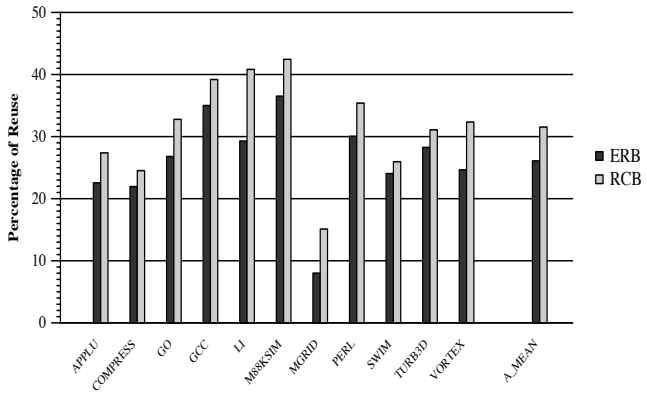

Figure 5. Enhanced Reuse Buffer vs. Redundant Computation Buffer for history depth of 1.

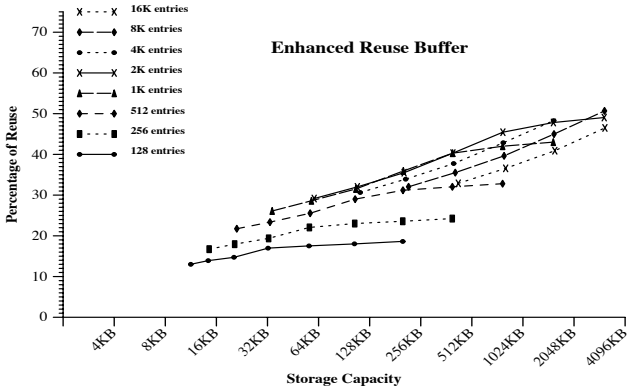

Figure 6. Reuse exploited by the Enhanced Reuse Buffer.

6.2 Basic Reuse Statistics

This subsection presents statistics about the percentage of reuse[1] that can be exploited by the different schemes for an ideal scenario in which all instructions are assumed to have their operands ready in the decode stage. The objective of this study is to investigate which is the best configuration for each reuse scheme and these best configurations will be later evaluated for a superscalar processor. In this latter scenario, the unavailability of the source operands will prevent the exploitation of reuse.

We first show that exploiting quasi-common subexpressions is a significant source of reuse. This is illustrated in Figure 5, which shows the percentage of dynamic instructions reused when the history depth is one and the Atable of each scheme has 1024 entries. The Mtable and the Vtable are assumed to have 512 and 1024 entries respectively in all the experiments. The total size of the tables for the experiment in Figure 5 is approximately the same: 34 KB (KiloBytes) and 36 KB for the Enhanced Reuse Buffer and the Redundant Computation Buffer respectively. Note that both schemes can reuse a significant number of instructions and RCB numbers are always higher than those of the ERB. On average, the reuse due to quasi-common subexpressions is about 6% and it is significant for all programs

Figure 6 shows the percentage of dynamic instructions reused by the Enhanced Reuse Buffer. Only average numbers (arithmetic mean) for the set of benchmarks are shown. The X axis shows the total storage capacity in KiloBytes (KB). Each line corresponds to a different number of entries in the Atable (the size of the Mtable is kept fixed and it is also considered in the total capacity). The different dots in a line correspond to different history depths, starting from 1 at the leftmost point and being increased by a factor of two from one point to the other.

The results for 128 and 1024 entries with history depth of one are consistent with the results presented in [11]. For instance, the 1024-entry configuration can reuse 26% of the dynamic instructions (25.7% was reported in [11]). Increasing the history depth provides a significant improvement as implied by the positive slope of the curves. For instance, for the 1024-entry configuration, the reuse grows from 26% to 41% when the history depth increases from 1 to 16. Note also that the benefits of increasing the history depth are more noticeable for a large number of entries. Looking at the best trade-off between number of entries

[1] For all figures, percentage of reuse is shown over all the dynamic instructions, where arithmetic operations, conditional branches, memory addresses and load values are the ones capable of being reused. Note that memory operations count twice (once for the address calculation and another for the memory value).

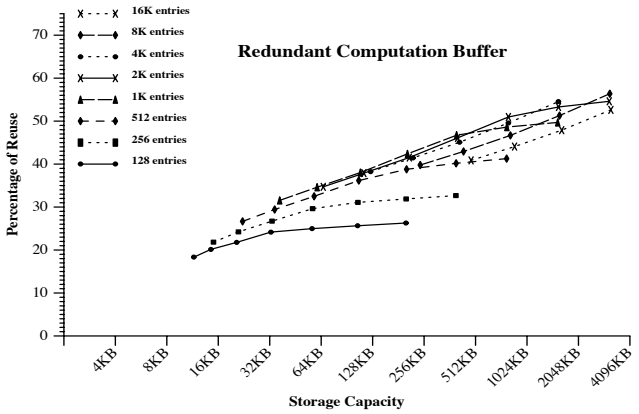

Figure 7. Reuse exploited by the RCB.

and history depth, we can conclude that for small capacities a history depth of one is the most effective configuration, whereas for large capacities is not. This is explained by the fact that for a small capacity the additional storage is best spent in reducing interferences (by increasing the number of entries), whereas interferences are very rare for large capacities. A similar trend can be observed for the RCB in Figure 7 (note again that the size of the Mtable and Vtable are kept fixed and they are considered in the total capacity).

Note that the concept of history depth does not make sense for the result cache since its entries are not associated to particular static instructions. In fact, the result cache can store multiple results of the same static instruction by placing each one in a different result cache entry.

Figure 8 compares the three reuse schemes in terms of percentage of reused instructions for different total storage requirements. For each scheme and capacity only the best configuration is displayed. Note that in terms of reusability, the Enhanced Result Cache is the most effective scheme, followed by the RCB and the Enhanced Reuse Buffer. The effectiveness of the result cache is explained by the fact that this scheme allows a given instruction to reuse results from any other instruction with the same opcode as the RCB does, but it does not spend any storage to store pointers, and it does not need to first establish pairs of producers-consumers before reusing among different static instructions. On the other hand, the reuse buffer can just reuse results from the same static instruction.

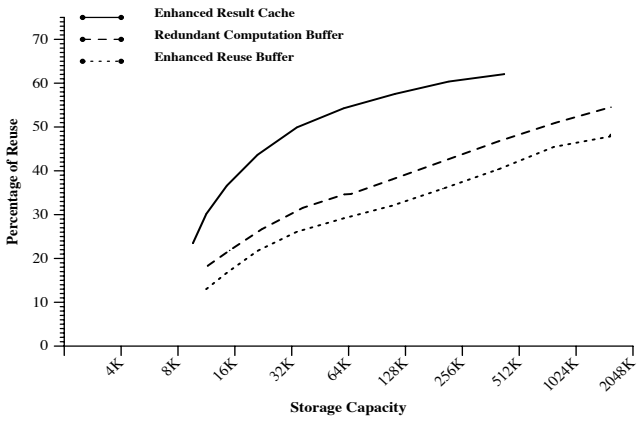

Figure 8. Reuse exploited by the best configurations of every scheme.

Instruction fetch	4 instructions per cycle.
Branch predictor	2048-entry bimodal predictor
Instruction cache	16 KB, direct-mapped, 32-byte cache line, 6-cycle miss latency
Instruction issue/ commit	Out-of-order issue, 4 instructions committed per cycle, 32-entry reorder buffer, loads execute only after all the preceding store addresses are known, store-load forwarding
Architected registers	32 integer and 32 FP
Functional units	4 integer ALUs, 2 load/store units, 4 FP adders, 1 integer mult/div, 1 FP mult/div
FU latency/repeat rate	integer ALU 1/1, load/store 1/1, integer mult 3/1, integer div 20/19, FP adder 2/1, FP mult 4/1, FP div 12/12
Data cache	16 KB, 2-way set-associative, 32-byte block, 6-cycle miss latency

Table 1: Parameters of the base microarchitecture.

6.3 Performance Figures for a Superscalar Microprocessor

This subsection presents performance figures of the different reuse schemes for a superscalar processor. Two different configurations are evaluated for each reuse scheme, one corresponding around 32KB and the other around 200KB of capacity[1]. For each capacity the best configuration has been chosen, according to the analysis presented in section 6.2. The base simulator models a 4-way dynamically-scheduled superscalar processor, based on the Register Update Unit [14], with the parameters shown in table 1.

Figure 9 shows the speedup achieved by the various reuse schemes for the base microarchitecture. The RCB scheme provides

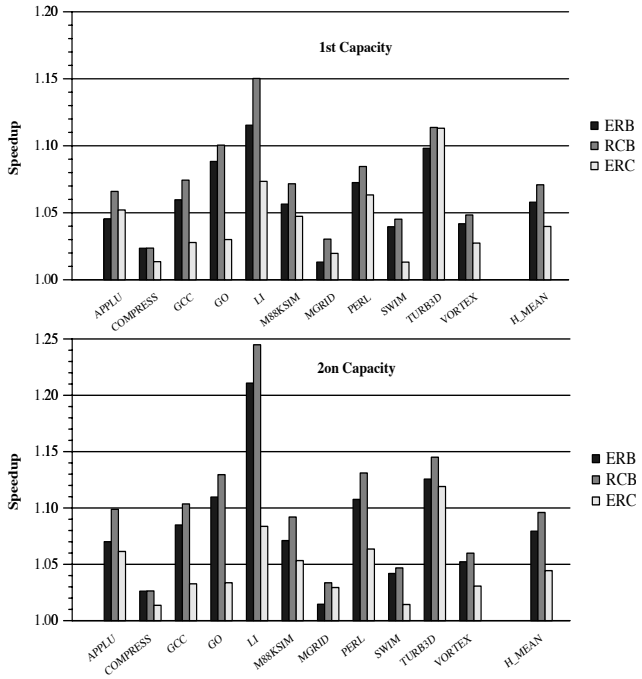

Figure 9. Speedup for the base microarchitecture for each program and the harmonic mean.

[1] Precise values are 34 KB, 36 KB, 34 KB and 217 KB, 221 KB, 217 KB for the Enhanced Reuse Buffer, Redundant Computation Buffer and Enhanced Result Cache respectively.

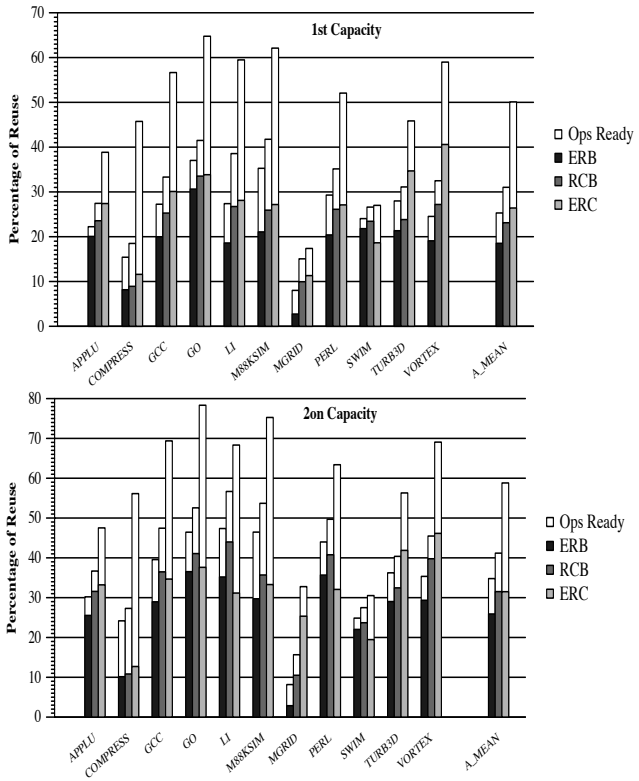

Figure 10. Reuse exploited in the superscalar

the highest speedup, whereas the result cache is the scheme with the lowest speedup, in spite of the fact that it is the scheme that exploits most reuse in the ideal case. However, the result cache is significantly penalized by not exploiting reuse chaining and by its higher reuse latency. It can be observed that the RCB provides significant speedups for all the programs. The highest benefit is experienced by *li*, which exhibits an speedup of 1.15 and 1.25 for the 32 KB and 200 KB configurations respectively.

Figure 10 shows the reuse exploited by the different schemes in the base microprocessor. It is also shown the reuse that each scheme exploits in the ideal case of always having all the operands ready. The actual reuse decreases when compared to the ideal reuse, despite the fact that the actual reuse includes some squash reuse [11] (reuse from squashed instructions due to a control misspeculation), which is not present in the ideal case. The result

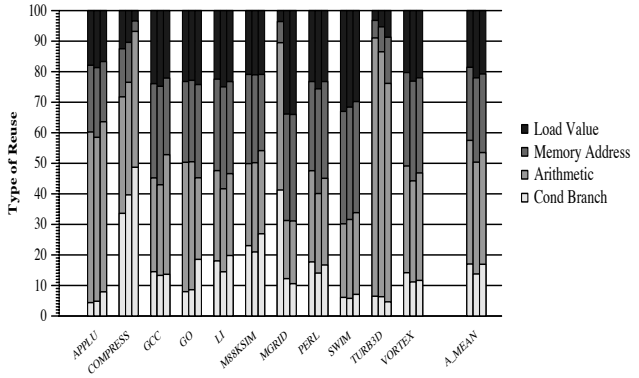

Figure 11. Contribution of each instruction category to total reuse. Each group of bars corresponds to ERB, RCB and ERC schemes respectively

cache is the scheme that loses most reuse when compared to the ideal case. An important reason for that is its inability to exploit reuse chaining. As a result of that, the result cache is still the scheme that exploits the highest amount of reuse but the difference against the RCB is much smaller than in the ideal scenario.

We have performed the same experiments for a more aggressive microarchitecture. We have assumed a 8-way issue superscalar processor with twice as many functional units as our base microarchitecture, and a more aggressive branch predictor based on an SAg [17] with 4096 branch history registers and a 4096-entry pattern history table. The average results are quite similar to those obtained for the base architecture, although there are some differences when looking at individual programs.

Figure 11 shows the contribution of each instruction category to the percentage of reuse shown in Figure 10. We observe that the contribution of each category to the total instruction reuse is similar and there are no significant differences when we compare among the different instruction reuse schemes.

7. A Hybrid RCB Scheme

The results in the previous section show that the result cache outperforms the RCB in terms of exploited reuse, but it provides a lower speedup due to its highest reuse latency. A hybrid RCB scheme could take benefit of the best of both approaches. The hybrid RCB scheme stores the result of each instruction in two different entries. One of the entries is indexed by the instruction address, as the RCB does, and the other is indexed by hashing the source operand values and the opcode, like the result cache. Note that the downside of this approach is that each dynamic instruction can use two entries of the buffer, and thereby it may produce more interferences.

When instructions are fetched, they are tried to be reused first

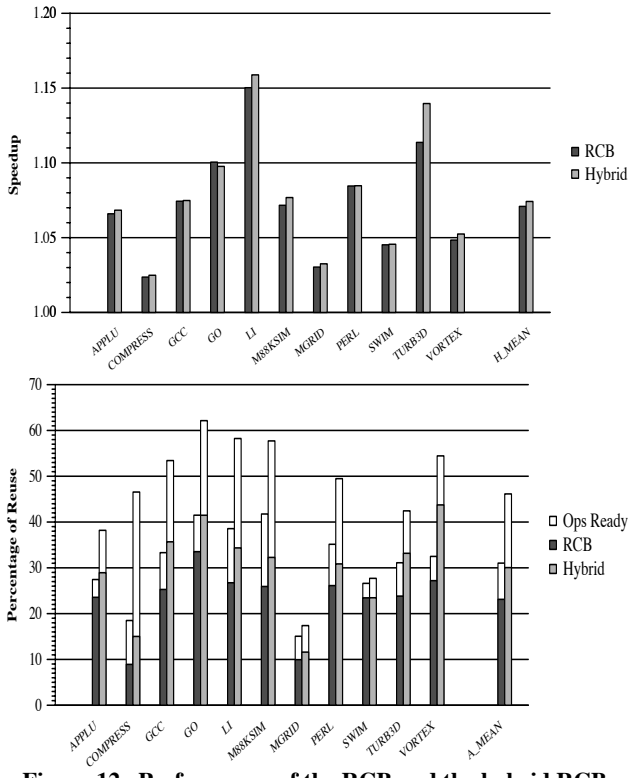

Figure 12. Performance of the RCB and the hybrid RCB scheme for a 32 KB capacity.

by indexing the Atable through the instruction address, and if this is not successful, they are tried to be reused by indexing it again through the hashed operand values and opcode. The first probe provides low latency whereas the second provides higher reusability.

Figure 11 compares the RCB and the hybrid RCB schemes for a 32 KB capacity (the results for 200 KB show a similar trend). Note that the hybrid scheme provides a slightly higher speedup and increases the percentage of reused instructions in about 7% (from 23% to 30%).

8. Performance Limits of Instruction-Level Reuse

We have finally evaluated the performance potential of instruction-level reuse for the base architecture. For this purpose we have assumed an ideal reuse engine that can reuse all the instructions (100% reuse hit for arithmetic instructions, loads, store addresses and conditional branches). The reuse latency is assumed to be the same as the RCB and reuse chaining is considered. The speedup by such an ideal reuse engine is shown in Figure 13. This figure shows that instruction-level reuse can provide a significantly higher benefits that those observed for the realistic schemes. This suggests that it may be worthwhile to investigate more powerful reuse engines.

9. Conclusions

We have presented in this paper a novel reuse mechanism that is referred to as *redundant computation buffer (RCB)*. This hardware mechanism can exploit reuse due to both quasi-invariants and quasi-common subexpressions, while it also exhibits a low reuse latency. When compared with previous schemes, which are extended with some novel features borrowed from the RCB, it exhibits the highest benefits in terms of execution time reduction, although it is not the scheme that achieves the highest reusability. The scheme with the highest reusability potential is penalized by being also the scheme with the highest reuse latency.

On average, the RCB can reuse around 30% of all dynamic instructions, which results in an 1.10 speedup. Improvements are experienced by all the programs, ranging from 3% for *compress* to 25% for *mgrid*. It has also been shown that storing multiple dynamic instances of the same static instruction may provide significant benefits, especially for large buffers.

A performance study of a perfect reuse engine shows that the performance potential of instruction-level reuse is high, and thus, the study of new reuse schemes may be an interesting area of research.

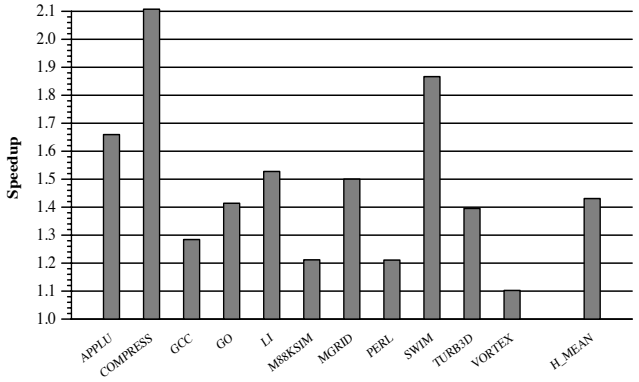

Figure 13. Speedup of a perfect reuse engine.

Acknowledgments

This work has been supported by the Spanish Ministry of Education under contract CYCIT TIC98-0511 and by the grant AP96-52460503. The research described in this paper has been developed using the resources of the European Center of Parallelism of Barcelona (CEPBA).

References

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

[2] M. Azam, P. Franzon and W. Liu, "Low Power Data Processing by Elimination of Redundant Computations", in *Procs. Int. Symp. on Low Power Electronics and Design*, pp. 259-264, 1997

[3] D. Burger, T.M. Austin and S. Bennet, "Evaluating Future Microprocessors: The SimpleScalar Tool Set". Technical Report CS-TR-96-1308. University of Wisconsin, July 1996

[4] A. González, J. Tubella and C. Molina, "The Performance Potential of Data Value Reuse", Technical Report UPC-DAC-1998-23, Universitat Politècnica de Catalunya, 1998

[5] S.H. Harbison, "An Architectural Alternative to Optimizing Compilers", in *Procs. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 57-65, 1982

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

[7] P. McNamee and M. Hall, "Developing a Tool for Memoizing Functions in C++", *ACM SIGPLAN Notices,* vol. 33, n. 8, pp. 17-22, Aug. 1998

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

[9] S. E. Richardson, "Exploiting Trivial and Redundant Computations", in *Procs. of Int. Symp. on Computer Arithmetic*, pp. 220-227, 1993

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

[11] A. Sodani and G.S. Sohi, "Dynamic Instruction Reuse", in *Procs. of Int. Symp. on Computer Architecture*, pp. 194-205, 1997

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

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

[14] G.S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers", *IEEE Transactions on Computers*, 39(3):349-359, 1990.

[15] S.P. Song, M. Denman and J. Chang, "The PowerPC 604 RISC Microprocessor", *IEEE Micro*, 14(5):8-17, Oct. 1994

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

[17] T. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction", In *Procs. of the 19th Annual International Symposium on Computer Architecture*, pp 124-134, 1992.