FACTA UNIVERSITATIS (NIŠ)
Series: Electronics and Energetics vol. 1(1995), 133–147

A SIMULATION STUDY OF THE "IGNORE" MECHANISM FOR DELAYED BRANCH CONTROL

Veljko Milutinović

Abstract. This paper is related to the experimental design of the RCA's 32-bit GaAs microprocessor on a single VLSI chip. The baseline architecture for this design was a Stanford University MIPS like machine which implements a software mechanism for the control of sequencing hazards. The RCA's GaAs architecture implements the same mechanism, but enhanced through the incorporation of the "ignore" instruction. This paper tries to provide an answer on how much does the incorporation of the "ignore" instruction speed up the execution of the compiled HLL code. The approach based on the "ignore" instruction is compared with both the SU-MIPS and the MIPS-X approaches to the problem, using both benchmarking and tracing.

1. Introduction

Stanford University MIPS architecture is well known [1], [20]. It uses a software mechanism for the control of sequencing hazards. At compile time, the branch delay slots are filled in using the algorithm of Gross and Hennessey [2], and the remaining slots are filled in with NOOPs. Advantages and disadvantages of this approach are discussed in a number of papers (e.g., [3]).

One of the disadvantages of this approach is that in relatively deep pipelines (as may be the case with pipelines of GaAs microprocessors) the percentage of NOOPs may increase drastically, which has a negative effect on the execution time of the code. This negative effect comes from the fact that the effective size of the cache memory becomes smaller when the percentage of NOOPs increases.

Manuscript received February 4, 1995.

This research was sponsored by RCA, and performed at Purdue University. A version of this paper was presented at the Purdue University Electrical Engineering Workshop on Computer Design, April 1985..

The author is with the Electrical Engineering Faculty, University of Belgrade, Bulevar Revolucije 73, 11000 Belgrade.

The problem can be briefly described in the following way. If the code contains no NOOPs, the percentage of loops that can fully fit into the cache of a given size is determined by the cache size (other factors also have an impact, but the impact of the cache size is of primary importance). For the same cache size, if the code contains NOOPs, the percentage of loops that can fully fit into the cache will decrease, and consequently the cache miss ratio will increase.

An important issue here is the penalty to be paid for each cache miss. The miss-related penalty is typically expressed in terms of the number of wasted clock cycles. Since this study is oriented to GaAs technology, the penalty for a cache miss may be fairly high (this depends on how advanced is the used packaging and interconnecting technology). Consequently, minimizing the number of NOOPs is of crucial importance.

There are a number of techniques for increasing the efficiency of branching. Some widely referenced papers on this subject include, but are not limited to [4], [5], [6], [7], [8], and [9]. However, the method implemented in the RCA's machine is different. As already indicated, it is based on the usage of the "ignore" instruction, and is described in [10].

2. The "Ignore" Instruction

The essence of the "ignore" instruction based solution is as follows. The "ignore" instruction is a separate instruction which is inserted into the branch delay slot, after the last useful instruction that is moved into that same slot, from elsewhere. The "ignore" instruction consists of the opcode field and the "ignore count" field. The value of the "ignore count" field determines how many cycles to follow are to be ignored. Therefore, the ignoring of the cycles to follow is done through a hardware mechanism rather than a software mechanism. Consequently, no NOOPs will be present in the code, and the cache hit ratio will not be affected in the above described way.

As far as the details, the solution to be studied here works as follows. The Gross/Hennessey algorithm is applied first, in the same way as in the architecture which fully relies on the software interlock. A certain number of branch delay slots will be filled in that way, and the "ignore" instruction (with a properly set "ignore count") is inserted next.

One drawback of the "ignore" instruction is that the cycle time of the microprocessor may have to be increased, due to the incorporation of the hardware control mechanism. However, the complexity of the "ignore" instruction related hardware control mechanism (especially in the case of the

RISC machines with a relatively simple state transition diagram) is much smaller compared to the complexity of the "conventional" hardware interlock mechanism. Therefore, the impact on the length of the clock cycle should be negligible, and is not included into this study.

3. Conditions of this Study

The architectures used in the first part of this study (benchmarking) are the standard Stanford University MIPS silicon architecture (architecture #1) and the RCA's GaAs architecture (architecture #2), both with and without the "ignore" instruction added to the instruction set. In the later part of the study (tracing) the MIPS-X architecture was assumed (architecture #3), as it is described in [19]. It was felt that such an approach would simplify the analysis without affecting its generality.

The code optimization algorithm used in this study is the above mentioned Gross/Hennessey algorithms, except for the difference that, in the case with the "ignore" instruction, the "ignore" instruction is inserted instead of the NOOP(s).

The choice of the code optimization algorithm definitely has an impact on the results of the analysis. This is simply because a more sophisticated code optimization algorithm would make the "NOOP case" less inferior, by being able to increase the amount of useful code migrated into the branch delay slots (which decreases the number of NOOPs in the code). For that reason, this study includes a part which provides the answer to a more general case in which the percentage of successfully filled branch delay slots can be arbitrarily modified.

The simulator of architecture #1, architecture #2, and architecture #3 is built using the ISP' simulation language and the ENDOT simulation tool [11]. Implementational structure and human interface of this simulator are of the "business as usual" style, and will not be described here. For more information on that subject, the interested reader is referred to [12].

After the results are extracted from the simulator, they are represented in a way which underlines the relative differences between the "ignore case" and the other two approaches. Since execution time is relevant here, the performance measure was chosen to be the ratio of execution times for the two cases being compared.

Finally, in order to make the results of this study more widely applicable, and also to address the algorithm choice related problem mentioned above, the study consists of two parts: benchmarking (explicit analysis and implicit analysis) and tracing. In the explicit analysis part of benchmarking, the values of the selected parameters are artificially varied so that a future designer can be made aware of the quantitative impacts of various code optimization algorithms. In the implicit part of benchmarking, the simulation is repeated for two relatively short "standard" benchmarks. More details about the chosen benchmarks can be found in [12, 13, and 18]. In the last part, two relatively long traces are used (about 4M instructions each one).

In conclusion, "explicit" benchmarking (with synthetic benchmarks) is used to obtain information about the impact of various design parameters, "implicit" benchmarking (with short semantic benchmarks) is used to obtain a rough feeling about a broader set of relevant aspects, and the tracing (with long semantic traces) is used to obtain a precise information about a narrow set of the most important aspects.

4. Processor Issues of Relevance

As already indicated, the benchmarking analysis to follow is done separately for the Silicon Stanford University MIPS and the GaAs RCA MIPS machines. Within each of these two mainstreams, the major issue of importance is the ratio of total cycles for the cases without the "ignore" instruction and with the "ignore" instruction. The tracing was done for the MIPS-X and the GaAs RCA MIPS machine, using a similar performance measure.

The instruction sets of the processors are defined in [1], [3] and [19]. The number of registers in each case is chosen to be 16. The depth of the pipeline is 5 for the Stanford University MIPS (new instruction scheduled on every other cycle), 3 for the MIPS-X, and 9 for the RCA GaAs MIPS. The cycle time was normalized, to enable a fair comparison.

The number of branch delay slots is related to the depth and the organization of the pipeline. However, the exact number of branch delay slots differs from instruction to instruction. For the silicon MIPS machines, it is 1 or 2 [1 and 19]. For the GaAs RCA MIPS machine, it is 5 [10]. Finally, in order to make the results easier to compare, the packing capability of the Stanford University MIPS (packing two operations into one instruction) was not exploited in this study.

5. Cache Memory Parameters of Interest for this Study

This work assumes separate instruction and data caches. The analysis to follow is related to the instruction cache, and is influenced in large part by the former work of A.J. Smith (e.g., [4] and [15]). The major issue of importance is the cache size (S) expressed in words. Values of the parameter S chosen for the benchmarking part of the study are:

$$S = \{32, 64, 128, 256, 512, 1K, 2K\}.$$

Reasons for choosing this set of values is multifold. First, the stress here is on GaAs technology, and GaAs caches tend to be small. Second, the lowest chosen value of 32, is below the nominal expected cache size for realistic implementations, and can be treated as the lower bound. Analogously, the value of 2K was chosen as the upper bound, based on the existing GaAs microprocessor implementations [14]. The second major issue of importance here is the number of extra cycles lost on the cache miss (M), due to the long delays of accessing the main memory. Values of parameter M chosen for this study are:

$$M = \{1, 2, 3, 4, 5, 6\}.$$

Reasons for choosing this set of values are also multifold. Typical value for Silicon designs is $M = 1$ (and will be used as a default value for Stanford University MIPS related studies to follow). Typical value for GaAs designs is $M = 3$ (and will be used as a default value for RCA MIPS related studies to follow). For GaAs, this assumes that the first level cache is on the same package as the CPU, and the second level cache, which is off the CPU package, is large enough. However, if the second level cache is not large enough, or it does not exist, the value of the parameter M can be as large as 6, or even 9 (this depends on the quality of interconnecting and packaging technology, and the size of main memory). Therefore, the chosen set of values for the parameter M covers nicely the entire range of practical values.

The major reason for selecting those two machines for the benchmarking part of the study is given in the introduction of this paper. Once the choice of architecture #1 and architecture #2 was made, it became apparent that the choice is also good for two more reasons. First, this choice gives the situations which are typical for Silicon and GaAs, the two major technologies these days. Second, this choice gives the situations which can conditionally be treated as lower and upper bounds.

Other cache related issues of importance include the level of associativity, replacement policy, main-memory updating, etc. [15]. These issues are not analyzed as a part of this study. For each of these issues one representative solution was chosen, and was kept "frozen" throughout the study.

As far as the level of associativity, the direct mapping was chosen. The reason for this decision is that most of the existing GaAs systems have chosen direct mapping [14]. As far as other relevant cache design issues, everything was done as in [10]. The main reason for selecting architecture #2 and architecture #3 for the tracing part of this study is because of the need to compare the "ignore" approach with a "state-of-the-art" approach.

6. Parameters of the Explicit Analysis

As indicated earlier, this type of analysis is based on predetermined statistics for relevant parameters of the application software and the system software.

This type of analysis helps to evaluate the performance of the future systems, over a wider range of application software characteristics, and for both current and future possibilities of the code optimizers.

As far as the application software, three issues are crucial here. First, the percentage of branch instructions. Second, the distribution of loop sizes. Third, the distribution of forward and backward branches.

Typically, the probability of branches (B) for different applications runs between 10% and 30% [14]. Consequently, the chosen range of values for the parameter B is:

$$B = \{10\%,\ 20\%,\ 30\%\}.$$

The default value to be used on the figures to follow is $B = 20\%$. The reason for this choice is that $B = 20\%$ represents the arithmetic average. Typical distribution of loop sizes is difficult to determine. One general way to treat the distribution of loop sizes is via a weighted sum of individual loop sizes. The average loop length is given by:

$$\overline{L} = \sum_{i=1}^{n} \omega_i l_i, \quad i, n - \text{integers},$$

where ω_i refers to the percentage of loops of the length l_i $(i = 1, 2, \dots, n)$. The software tool developed for this study is able to work with various values of ω_i and l_i $(i = 1, 2, \dots, n)$. However, the results presented here imply a uniform distribution of loops of the sizes 4 to 2K. In other words, 10% of the loops are of the length 4 words, 10% of the loops are of the length 8, 10% of the loops are of the length 16, etc. This seemed to be a fair approximation of several benchmarks that were crucial for the application of the RCA's GaAs microprocessor.

Typical ratio of forward and backward branches is also difficult to determine. In the absence of a better ratio, it was decided that the results are represented only for the 1 : 1 ratio of forward and backward branches. In the first approximation, this refers to the percentage of branches related to loops, versus the branches not related to loops. The chosen ratio matched fairly well the typical situation in benchmarks that were crucial for the application of the RCA's GaAs microprocessor.

As far as the system software, the major parameter of importance is the fill-in capability (F) of the optimizing compiler. It tells how many branch delay slots have been filled in with useful instructions. The range of values chosen for this analysis is:

$$F = \{1, 2, 3, 4\}.$$

Such a range was chosen to accommodate the needs of the RCA MIPS architecture. For the Stanford University MIPS architecture, the values of F above $F = 2$ will give the same results. Consequently, the default value for some of the figures to follow will be $F = 2$. Finally, it was assumed that the instruction cache was empty, initially. Once again, the definition of the "performance ratio" used on the plots later in this paper is as follows: Ratio of the number of execution cycles with the "ignore" instruction in the branch delay slots, and the number of execution cycles with NOOPs in the branch delay slots. The maximum value of the performance ratio is equal to 1. The lower the value of the performance ratio, the better (relatively) the performance of the "ignore" instruction-based system.

7. Parameters of the Implicit Analysis

As indicated above, this type of analysis is based on the selected benchmarks. Values of relevant parameters are hidden into the benchmarks. This type of analysis helps to get a better feeling about the performance for the actual code, using a specific code optimization algorithm (all results to follow imply the optimized code).

As far as the application software, here we present the results for two benchmarks provided to Purdue University by RCA. For more information on these benchmarks, the interested reader is referred to [13 and 18]. In some way, these two benchmarks represent two extremes, when it comes to the percentage of branches.

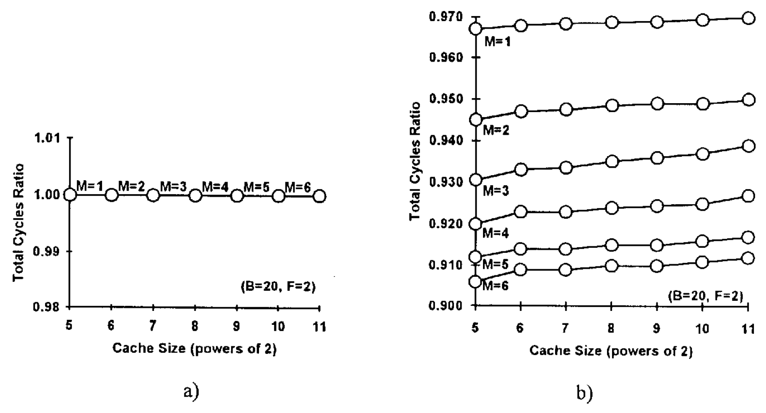


Fig. 1. Total cycles ratio vs. cache size, varying extra cycles lost on a cache miss: (a) Stanford MIPS, (b) RCA GaAs.

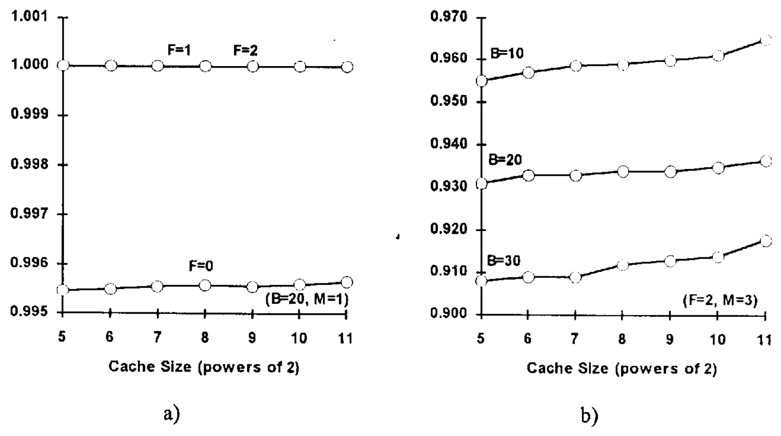


Fig. 2. Total cycles ratio vs. cache size varying percentage of branch instructions: (a) Stanford MIPS, (b) RCA GaAs.

As indicated in [12], the first benchmark, intmm, is an integer matrix multiplication program. This benchmark initializes two 40*40 integer matrices and multiplies them. The second benchmark, search, is an adaptation of a Motorola MC68020 string search benchmark [18]. While the MC68020 benchmark was set up to search a block of test data for the first occurrence of a given byte string, the adaptation used searched the test data block for a given word string. The test data used was a block of 120 words and the pattern searched for was 15 words long. A partial match, the first four words, was imbedded in the test block starting with the 61st word, while the complete match began with the 91st test block word [12]. As already indicated, the two benchmarks were chosen to represent opposite ends of the regime in terms of percentage of branch instructions. Neither program utilized loops in multiplication or division algorithms; the difference in branch percentages was due to the different methods of implementing multiplication and division in the two processors. Approximately 37% of the useful (i.e., neither NOOP nor "ignore") instructions executed for the search program were branches [12].

8. Performance Diagrams of Interest for this Study

Careful selection of simulation outputs is of large importance for good understanding of simulation results. Figures 1 to 4 to follow have the following characteristics:

1. X axis will refer to the size of the cache (S).
2. Y axis will refer to the ratio of the number of cycles without and with the "ignore" instruction (R).

The choice of S reflects the fact that the major impact of the "ignore" instruction is through the changed effective value of the cache size. The choice of R reflects the fact that the major goal of this study is to determine the actual speed-up due to the utilization of the "ignore" instruction.

Figures 1 to 4 were obtained using benchmarks. The major rationale behind this benchmarking (with relatively short benchmarks) was to get a rough feeling about the efficiency of the "ignore" instruction.

Figures 5 and 6 were obtained using traces. The major rationale behind this tracing (with relatively long traces) was to get a precise comparison between the "ignore" instruction and the delayed branch with squashing used in the MIPS-X processor [19], which is a superior solution to the problem in typical silicon environments. The performance measure RR was selected so that the two approaches can be compared easily, and consequently is different from the performance measure R used in benchmarking.

It is believed that the chosen figures shed the complete light on the problem. As indicated below, other functional dependencies of interest can be easily created with the software tool that was built for this study. For more details on the tool, its structure, as well as data testing and verification procedures, the interested reader is referred to [12].

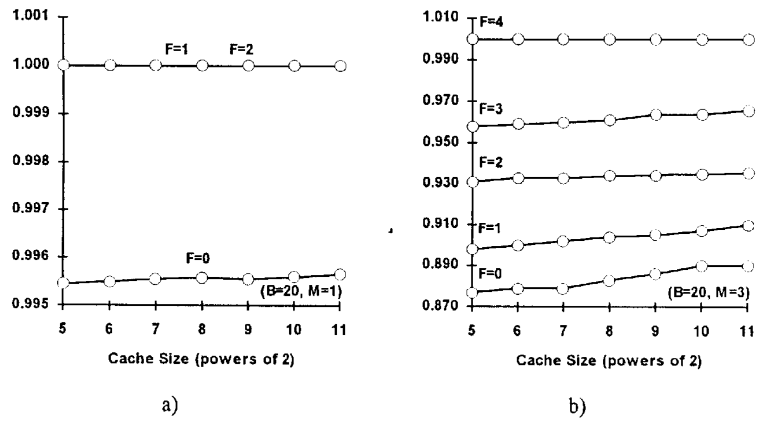


Fig. 3. Total cycles ratio vs. cache size, varying branch delay slot fill-in: (a) Stanford MIPS, (b) RCA GaAs.

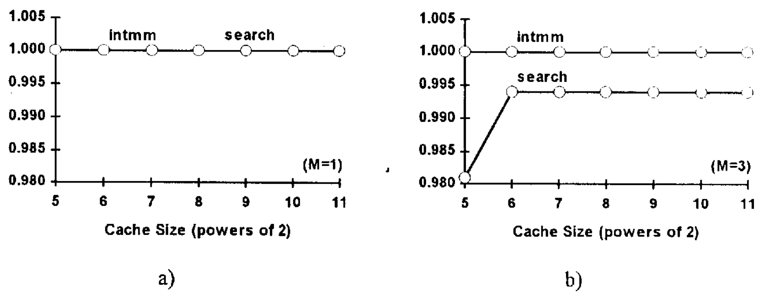


Fig. 4. Total cycles ratio vs. cache size for benchmark programs: (a) Stanford MIPS, (b) RCA GaAs.

9. Discussion of Results for the Explicit Analysis

Figure 1 compares Stanford MIPS (Figure 1a) and RCA GaAs machine (Figure 1b) for various values of parameter M. Obviously, as indicated in Figure 1a, parameter M has no impact in the case of the Stanford MIPS (shallow) pipeline structure, since the default value of parameter F was chosen to be $F = 2$, to reflect the situation typical of state-of-the-art code optimizers. On the other hand, in the case of the RCA GaAs machine (deep) pipeline structure, for the same type of code optimization technology ($F = 2$), the usage of the "ignore" instruction can provide an improvement of up to about 10%. I believe that this level of improvement fully justifies the incorporation of the "ignore" instruction into the RCA GaAs machine [10]. As you remember, the primary motivation for this study was to provide a justification for including (or not including) the "ignore" instruction.

Figure 2 compares Stanford MIPS (Figure 2a) and RCA GaAs machine (Figure 2b) for various values of parameter B. Again, due to the fact that the value of F was chosen to be $F = 2$, the impact of parameter B is not visible (the plot was included for consistency reasons, and also to show clearly the differences between a "shallow" pipeline and a "deep" pipeline). However, in the case of the RCA GaAs pipeline, for the chosen values of F and M ($F = 2$ and $M = 3$), the performance gain was about 2% to 3%, for every 10% increase in the percentage of branches (B).

Figure 3 compares Stanford MIPS (Figure 3a) and RCA GaAs machine (Figure 3b) for various values of parameter F. In the case of $F = 0$, the "ignore" instruction can provide some improvement even in the case of the Stanford MIPS pipeline ($F = 0$ means no code optimization). However, that improvement is only about 1% (negligible). On the other hand, the $F = 0$ case provides an improvement of about 12% for the RCA GaAs machine (upper bound). Note that the maximum percentage of branch instructions is defined by:

$$B_{max} = \frac{100}{F+1}.$$

As indicated in [12], in the context of this analysis, each of the branch instructions was treated independently. In other words, the branches were inserted randomly to yield the specified branch instruction percentage and each was assigned a random branch size and direction as per the assumptions stated previously. Thus, each branch was assumed to be a taken branch and there was no provision for simulating repeated loops. This may have the effect of producing more "scattered" code than might normally be encountered. As a result, the simulation output may produce more cache misses for a given cache size than might be experienced using actual compiled code. The effect this would have on the results presented is not apparent, as the total cycles ratio is dependent on the difference in cache misses between using NOOPs and using the "ignore" instruction, not the actual number of cache misses. Load delay slot fill-in was not addressed in the simulation. It was assumed that all load delay slots were filled in with useful instructions. This is unrealistic for the RCA GaAs architecture, which has eight load delay slots to fill. As a result, the simulation output would tend to produce less cache misses than would actual compiled code, but again, it is difficult to predict what effect this would have on the total cycles ratio. An "ignore" instruction approach to load delay slots could decrease execution time for architectures with deep pipelines where it is likely that the compiler would not be able to fill two or more load slots [12].

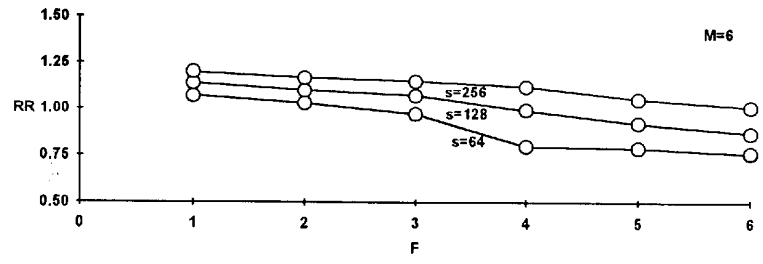
10. Discussion of Results for the Implicit Analysis

Figure 4 gives results of the implicit analysis for Stanford MIPS (Figure 4a) and RCA GaAs machine (Figure 4b). For the Gross/Hennessy code optimization algorithm (parameter F is irrelevant now), which fills in all (or all minus one) branch delay slots, in both pipelines, for the benchmark intmm, the "ignore" instruction provides no improvement. However, it does provide a minimal improvement (only about 1%) for the other extreme, the benchmark search. This result was not very encouraging as far as the inclusion of the "ignore" instruction into the instruction set of the RCA GaAs machine. In other words, the value of the "ignore" instruction is highly dependent on the end application and the choice of the code optimization algorithm.

As also indicated in [12], the advantage of using the "ignore" instruction is rather unclear in Figure 4, since the size of the benchmark program is small compared to the cache sizes used. The NOOP version of the RCA search program occupied only 68 words and the "ignore" version occupied only 47 words. As a result, no change in the total cycles ratio would be expected for any cache size greater than 68 words. The only difference expected in that case would be the extra cycles for a larger number of cache misses encountered when initially loading the larger NOOP version of the program into the instruction cache [12].

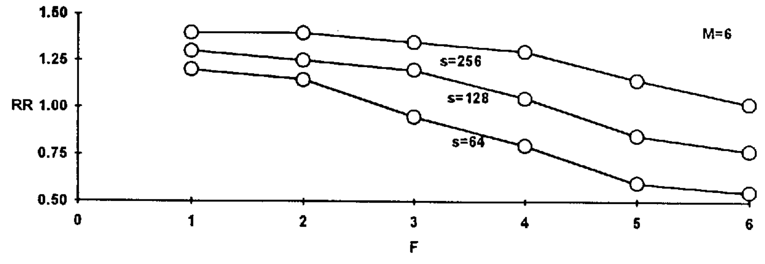
11. Discussion of Results for the Tracing

Figure 5 is related to a trace which is intensive in numerical computing, and Figure 6 is related to a trace which is intensive in symbolic computing. The first trace includes extensive matrix multiplication and fft operations. The second trace includes code which is related to artificial neural networks (ann). Percentage of branches in the two traces is different, and that explains some numerical differences between Figures 5 and 6.



RR- N umber of cycles for the trace with "ignore" instruction divided by the number of cycles for the trace with delayed branch with squashing;
S- C ache size;
F- T he value of the fill-in parameter.

Fig. 5. Results of tracing with a 4M instruction trace oriented to numerical computing (intensive in matrix multiplication and fft): "Ignore" instruction versus delayed branch with squashing.



RR- Number of cycles for the trace with "ignore" instruction divided by the number of cycles for the trace with delayed branch with squashing;
S- Cache size;
F- The value of the fill-in parameter.

Fig. 6. Results of tracing with a 4M instruction trace oriented to symbolic computing (intensive in Hopfield and other ann code): "Ignore" instruction versus delayed branch with squashing.

Performance measure RR on Figures 5 and 6 represents the ratio of the number of cycles needed to execute a trace with the "ignore" instruction, and the delayed branch with squashing. If RR ¡ 1 then "ignore" instruction provides better performance than the delayed branch with squashing.

For different cache sizes, the "ignore" instruction performs better for the value of F parameter equal to 4 or 5. This means that the approach based on the "ignore" instruction is better for deeper GaAs pipelines, and not too attractive for typical silicon pipelines. Also, Figures 5 and 6 show that the approach based on the "ignore" instruction compares better for smaller instruction caches, which is another characteristic of GaAs systems.

12. Conclusion

Usage of the "ignore" instruction (with an associated hardware mechanism for ignoring the given number of cycles) can speed up program execution as compared to using NOOPs (in unfilled branch delay slots), in situations where more than one branch delay slot per branch is left unfilled by the code optimizer. For architectures such as the Stanford University MIPS (with only one or two branch delay slots to fill), only marginal improvement is realized by incorporating the "ignore" instruction. For architectures with deeper pipelines and more branch delay slots (such as the RCA GaAs architecture), the use of the "ignore" instruction can lead to a decrease in execution time, even when compared to delayed branch with squashing, which justifies the usage of the "ignore" instruction in [10]. This paper provides a numerical answer to the question of how significant is the decrease in execution time, and it complements the earlier paper [10].

Acknowledgment

Initial request for this study came from Walter Helbig of RCA (now GE Advanced Technology Laboratories). The author is thankful to Greg Dunn of Purdue University for his help in the simulation domain. Results and conclusions from his simulator are used directly in this paper. The author is also thankful to Dee Dee Dexter for the help in typing and editing of this paper.

REFERENCES

1. Przybylski, S. A., Gross, T. R., Hennessy, J. L., Jouppi, N. P., Rowen, C.: *Organization and VLSI Implementation of MIPS.* Stanford University Technical Report, Palo Alto, California, U.S.A., No. 84–259, April 1984.
2. Hennessy, J. L., Gross, T. R.: *Optimizing Delayed Branches.* Proceedings of the MICRO-15 Workshop, Palo Alto, California, U.S.A., October 1982.
3. Milutinović, V., Editor: *High-Level Language Computer Architecture.* W. H. Freeman/Computer Science Press, New York, New York, U.S.A., 1989.

4. Lee, J. K. F., Smith, A. J.: *Branch Prediction Strategies and Branch Target Buffer Design.* IEEE Computer, January 1989, pp. 6–22.
5. McFarling S., Hennessy, J.: *Reducing the Cost of Branches.* Proceedings of 13th Annual International Symposium on Computer Architecture, Tokyo, Japan, June 1986, pp. 396–403.
6. DeRosa, J. A., Levy, H. M.: *An Evaluation of Branch Architectures.* Proceedings of the 14th Annual International Symposium on Computer Architecture, Pittsburgh, Pennsylvania, U.S.A., June 1987, pp. 10–16.
7. Ditzel, D. R., McLellan, H. R.: *Branch Folding in the CRISP Microprocessor.* Proceedings of the 14th Annual International Symposium on Computer Architecture, Pittsburgh, Pennsylvania, U.S.A., June 1987, pp. 2–9.
8. Cartadella, J., Llaberia, J. M.: *Zero-Delay Branches in a RISC.* Internal Report, University of Barcelona, Barcelona, Catalonia, Spain, 1987.
9. Lilja, D. J.: *Reducing the Branch Penalty in Pipelined Processors.* IEEE Computer, July 1988, pp. 47–55.
10. Helbig, W., Milutinović, V.: *A DCFL E/D-MESFET GaAs Experimental RISC Machine.* IEEE Transactions on Computers, February 1989, pp. 263–174.
11. Rose, C. W., Ordy, G. M., Drongowski, P. J.: *N.mPc: A Study in University-Industry Technology Transfer.* IEEE Design and Test, February 1982, pp. 44–56.
12. Dunn, G.: *Simulator for the Analysis of Delayed Branch Architectures.* Purdue University Internal Report, West Lafayette, Indiana, U.S.A., August 1989.
13. Fura, D.: *Architectural Approaches for GaAs Exploitation in High-Speed Computer Design.* Purdue University Technical Report, No. TR-EE 85–17, December 1985.
14. Milutinović, V., Editor: *Microprocessor Design for GaAs Technology.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1990.
15. Smith, A. J.: *Cache Memories.* ACM Computing Surveys, September 1982, pp. 473–530.
16. Gill, J., Gross, T., Hennessy, J., Jouppi, N., Przybylski, S., Rowen, C.: *Summary of MIPS Instructions.* Stanford University Technical Note, No. 83–237, Palo Alto, California, U.S.A., November 1983.
17. Chow, P., Geigel, T., Milutinović, V., Pridmore, J.: *Impact of Mapping Parameters on the Performance of Small Cache Memories.* Microprocessors and Microsystems, Vol. 12, No. 4, May 1988, pp. 197–205.
18. Ripps, D., Mushinsky, B.: *Benchmarks Contrast 68020 Cache-Memory Operations.* EDN, Vol. 30, No. 18, 8. August 1985, pp. 177–202.
19. Chow, P.: *MIPS-X.* Kluver Publishing Company, Boston, Massachusetts, U.S.A., 1989.
20. LR3000 and LR3000a MIPS RISC Microprocessor: *User's Manual.* LSI Logic, Milpitas, California, U.S.A., 1990.