*** RELATÓRIO TÉNICO ***

AVALIAÇÃO DE UMA ARQUITETURA SPARC COM CACHE DE DESVIO E BARRAMENTO TIPO HARVARD

Gabriel Pereira da Silva
Júlio Salek Aude

NCE 18/91
Outubro/91

Universidade Federal do Rio de Janeiro
Núcleo de Computação Eletrônica
Caixa Postal 2324
20001 - Rio de Janeiro - RJ
BRASIL

Este artigo foi apresentado na EUROMICRO'91, realizada em Viena/Áustria, de 02 a 05 de setembro de 1991.

AVALIAÇÃO DE UMA ARQUITETURA SPARC COM CACHE DE DESVIO E BARRAMENTO TIPO HARVARD.

Resumo

Variações na arquitetura SPARC são estudadas neste artigo, com particular enfase no uso de uma cache de desvio e de um barramento Harvard. Um simulador que funciona em um modo ciclo a ciclo foi desenvolvido para realizar medidas de desempenho em várias configurações. Os resultados obtidos são apresentados neste artigo.

EVALUATION OF A SPARC ARCHITECTURE WITH HARVARD BUS AND BRANCH TARGET CACHE

Abstract

Variations in the SPARC architecture are studied in this paper, with particular emphasis to the use of a branch target cache and a Harvard bus. A simulator that works in a cycle per cycle basis has been developed to conduct performance measurements of some configurations. The results obtained are reported in this paper.

EVALUATION OF A SPARC ARCHITECTURE WITH HARVARD BUS AND BRANCH TARGET CACHE

Gabriel P. Silva and Julio S. Aude

N.C.E., Federal University of Rio de Janeiro
P.O Box 2324, Cep 20001, Rio de Janeiro, Brasil.
Tel:55-21-5983212 Fax: 55-21-2708554 E-mail: ncd01010@ufrj.bitnet

The SPARC architecture is studied in this paper, with particular emphasis to the use of a branch target cache and a Harvard bus. Performance measurement with some configurations are reported. A simulator that works in a cycle per cycle basis has been developed to conduct those measurements.

1. INTRODUCTION

The SPARC was defined by Sun Microsystems in 1987 [1,2] based on Berkeley's RISC architectures. It is an open architecture with a high degree of architectural freedom that allows specific hardware implementations while keeping binary compatibility. This paper describes the results obtained by a SPARC simulator that makes an evaluation of different SPARC configurations.

The SPARC is a 32 bit RISC architecture, based on RISC I/II and SOAR Berkeley designs. It has two main units: a Floating Point Unit and an Integer Unit. Each of these units has its own register file. The Integer Unit may have between 40 and 520 registers, depending on the implementation. These registers are partitioned into 2 to 32 overlapping windows, that contain parameters passed between routines. This organization allows around 10 to 20% fewer memory references than stack organized architectures[2].

The SPARC is a pipelined architecture, and most transfer control instructions are one slot delayed, that means, the instruction immediately following the jump instruction is also executed. Some other features are not originally specified in the architecture's definition. They include the size of the bus and its organization, the use of a internal cache, number of pipeline stages, the type of control logic (RANDOM/PLA) and technology of implementation (ECL, CMOS).

The simulator explores configurations that result in implementations with higher performance, although with low hardware complexity. The parameters that the simulator evaluates are the register file size, the bus interface and the use of an internal branch target cache. With the enhancements proposed here, an architecture with an execution ratio as low as one cycle per instruction can be achieved.

2. THE OPTIONS EVALUATED

The register file size has been evaluated to obtain a minimum size, freeing chip area to implement other features. Specific routines to handle register window overflow/underflow have been assembled, creating a more realistic environment since their cost is also considered in program execution. The best register file managing strategy consists of moving one frame every time an underflow/overflow occurs, as indicated by Tamir and Sequin [3], and is used in the simulation.

The branch target scheme implemented was defined by Cortadella & Jové as Zero Time delay Branch [4]. It is intended for pipelined architectures and allows the evaluation of the conditional branch in parallel with the delayed slot instruction execution. The simulator assigns cache sizes in the range from 8 to 128 entries, each one composed of an instruction, target address and condition codes. The line size varies from 1 to 4 sets. The model simulated considers the use of an external memory with zero wait state instruction delivery.

The use of a Harvard bus organization, with independent data and instruction paths, is also

simulated. This was done because loads and stores account for 20% of the instructions executed, and data accesses compete with instruction fetches in a shared bus. With the use of separate data and instructions paths it is expected to get a considerable performance gain.

The simulated architectures differ among them in some points. While a shared bus has only four pipeline stages, the Harvard bus needs an extra pipeline stage. Also, the use of a branch target cache demand some modifications in the control of the bus fetch unit. These modifications were done in the simulator, but keeping compatibility with SPARC's definition.

The need for precise results concerning the microprocessor's behavior required the simulation of its operation on a cycle per cycle basis. To achieve this a detailed study of each instruction was done, with the definition of each pipeline stage operation. The trap instructions have been also emulated, and whenever a window underflow/overflow occurs, the program deviates to specific management routines. These routines have been hand coded and are loaded with the programs, acting as a little kernel, that saves and restores the window frames to/from simulator's memory.

The simulator does not allow I/O operations and does not simulate floating point operations. Also, it does not predict the cost related to context switching, which can happen in a real environment, though it can be estimated by the average number of activated windows.

3. THE BENCHMARK PROGRAMS

Conventional benchmark programs have been used in the evaluation. Those programs were written in C language [5,6] and some of them had been already used in other evaluations [7]. The code was compiled in a SPARCstation and then ported and adapted to the simulator.

The Quicksort program sorts a 1 Kword (4 Kbytes) array randomly filled, with recursive calls of best case O(N log N). The Sieve program generates series of prime numbers manipulating an array of 16 Kbytes. The Fibonacci program calculates the 18th member of a Fibonacci series through recursive calls. The Dhrystone program is a synthetic benchmark [6] with a mix of instructions of a typical integer application. In the code used there are only 256 iterations in the main program loop. The program Towers, that implements the Towers of Hanoi algorithm with 12 dishes, was used for register sizing simulation.

The number of simulated cycles of a program considers the cycles spent in save/restoring the windows from/to memory. The results of the simulation are expressed in terms of: number of cycles, number of instructions executed, cycles per instruction, average number of activated windows, data and instruction accesses, number of interlocks and instruction utilization.

4. RESULTS

Some of the results are summarized in figures 1 to 6. Figures 1 and 2 show the effects of register file sizing in three programs that use recursive calls. The "window miss ratio" in figure 1 expresses the ratio of save/restore instructions executed that don't find a window frame available in the register file. Figure 2 shows the total number of executed cycles for each program as a function of the register file size.

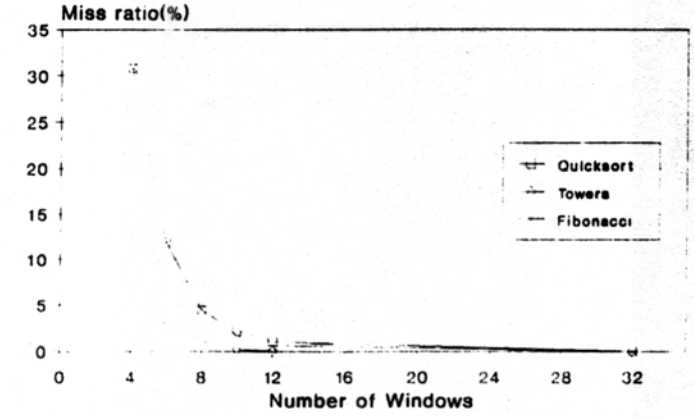
Figure 3 and 4 illustrate the influence of the branch target cache in the execution of Dhrystone program. The hit ratio of figure 3 expresses the ratio of conditional branch instructions that are found in the cache. Even for a cache with a total of 32 entries, hit ratios of 80% can be found, giving a considerable improvement in overall performance. The impact of this feature in the execution time of this program is shown in Figure 4.

Figure 5 shows the speedup obtained with the use of a Harvard bus organization, with independent paths for instruction and data. This analysis was done without the use of a branch target cache and with the register file configured with 8 windows. Figure 6 shows the speedup obtained if an internal branch cache with 64 positions is also added to the architecture. In this graphic, the average cycle per instruction execution ratio that is obtained with a "standard" and the proposed "enhanced" configuration is plotted.

5. CONCLUSIONS

The results obtained from the simulation lead us to the following conclusions:

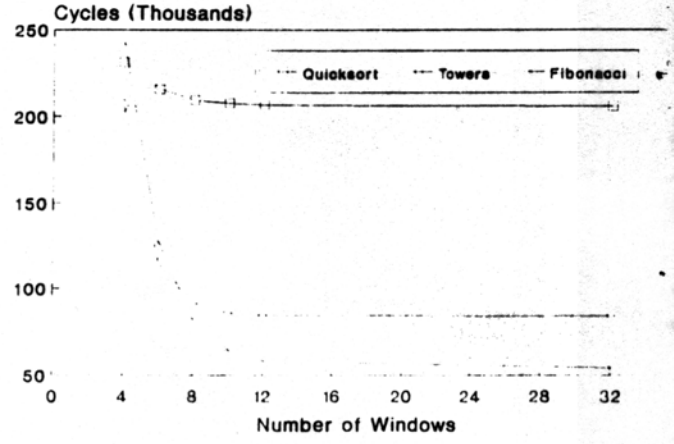
Register File Size



· fixed(1,1) strategy
·· miss ratio = traps / (saves+restores)

Figure 1

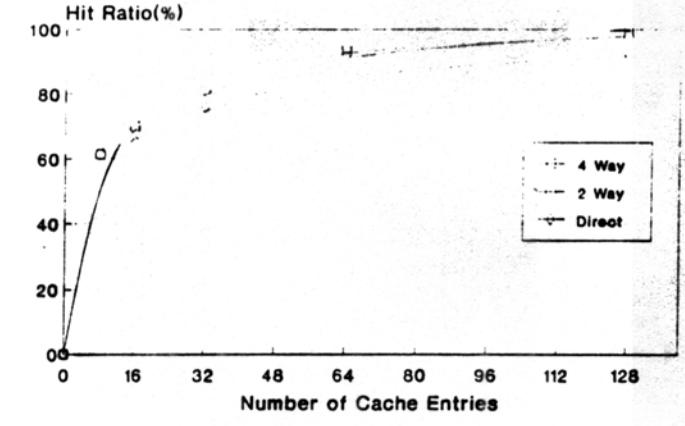
Register File Size



· fixed(1,1) strategy

Figure 2

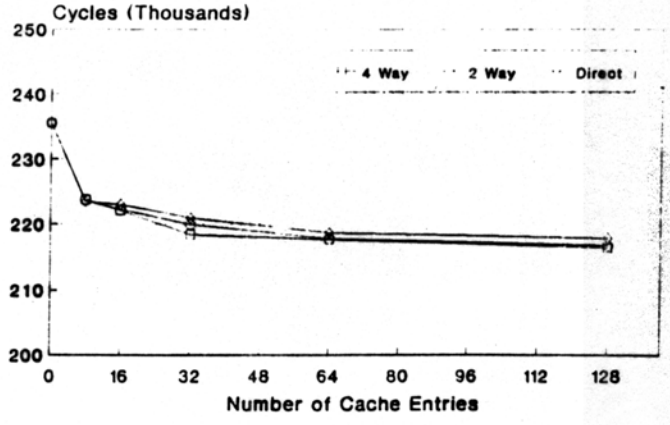
Branch Target Cache



· register file size = 8 windows
·· dhrystone

Figure 3

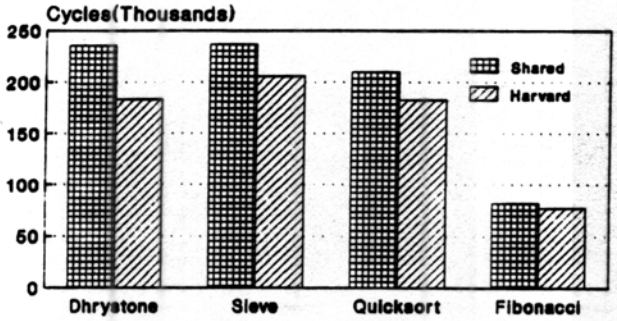
Branch Target Cache



· register file size = 8 windows
·· dhrystone

Figure 4

Bus Organization

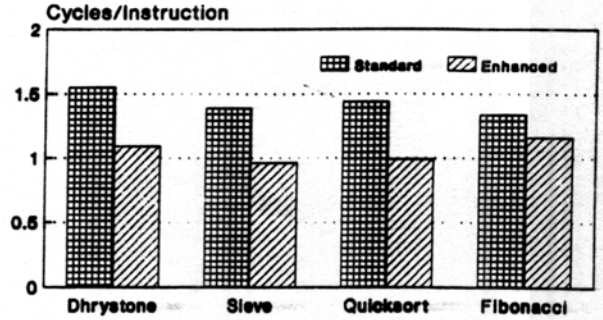


	Dhrystone	Sieve	Quicksort	Fibonacci
Shared	235.537	236.806	209.546	81.992
Harvard	183.108	205.425	182.683	77.352
Speedup	1.29	1.15	1.15	1.06

· without cache
·· register file size = 8 windows

Figure 5

Architecture Comparision



	Dhrystone	Sieve	Quicksort	Fibonacci
Standard	1.55	1.39	1.44	1.34
Enhanced	1.09	0.96	0.99	1.16
Speedup	1.42	1.45	1.46	1.16

· enh.: 64 entries, harvard bus
·· registerfile size = 8 windows

Figure 6

- The use of a register file with between 6 and 10 windows is optimal. The remaining area can be used to others features, such as an internal branch target cache.

- The traces indicate that caches of small sizes, with only 32 to 128 positions, are very effective. The set size seems not to be of significant influence, as also shown by Lee & Smith [8].

- The traces also shown that the use of a Harvard bus organization results in good improvement. It is problably the most expensive solution involved, but it is important if intensive data manipulating programs will be used.

- The association of a branch target cache with a Harvard bus can lead to execution ratios as low as one cycle per instruction.

This work has been used as a guideline to an implementation of a SPARC architecture that is currently being developed at NCE/UFRJ. Cost considerations are strongly dependent on the technology being used and are not done here. As a target for future work we intend to expand this simulator for handling superscalar architectures and study the data dependencies that can arise in this situation.

ACKNOWLEDGEMENTS

The authors would like to acknowledge CNPQ and FINEP, Brazil, for the support given to this research work.

* SPARC is a registered trademark of SPARC International, Inc.

REFERENCES

[1] Sun Microsystems Inc "The SPARC Architecture Manual", Mountain View CA, 1987. 199 pp.

[2] Garner, B. et all "The Scalable Processor Architecture" Proceedings of the IEEE COMPCON 88, New York, NY, IEEE, pp 278-283, 1988.

[3]Tamir, Y. & Sequin, C.H. "Strategies for Managing the Register File in RISC" IEEE Transactions on Computers, Vol C-32 (11): 977-989, Nov 1983

[4] Cortadella, J. & Jové, T. "Designing a Branch Target Buffer for Executing Branches with Zero Time Cost in a RISC Processor", Microprocessing and Microprogramming, North-Holland, 24: 573-580, 1988.

[5] Hinnant, D.F. "Benchmarking UNIX Systems" Byte, Peterborough, N.H., McGrawHill, 9(8):132-5,400-9, Aug 1984.

[6] Reinhold, P.W. "Dhrystone: A synthetic Systems Programming Benchmark", Communications of the ACM, New York, 27(10):1013-1030, Oct 1984

[7] Tamir, Y. "Simulation and Performance Evaluation of the RISC Architecture" Berkeley , CA, University of California, Memorandum, UCB/ERLM 81/17, 1981, 29 pp.

[8] Lee, J.K.F. & Smith, A.J. "Branch Prediction Strategies and Branch Target Buffer Design" Computer, New York, IEEE, 17(1):6-22, Jan 1984.