Cache Memory for a LISP Machine

Feitosa, R. Q., Guidacci da Silveira, G. F.
Catholic University of Rio de Janeiro
Electrical Engineering Department
R. Marquês de São Vicente, 225
22 453 - Rio de Janeiro - RJ; Brazil
e-mail: RAUL@GSC.ELE.PUC-RIO.BR

Abstract

This work is part of a research to find out the appropriate cache strategies for LISP oriented archictectures. A virtual LISP machine, based on the SECD architecture, was developed. Trace driven simulation was the used methodology to evaluate different cache structures. This paper analyses how three aspects of the SECD architecture affect the cache performance: free list organization, the allocation of the cons cells and garbage collection. The simulation results show that linear free lists and a unique contiguous vector of cons cell enforce access locality. The garbage collection does not have a cumulative effect over the cache miss ratio. The analysis is based only on traces that do not contain intervening garbage collections. The cache miss ratio for a given test program varies within a limited range, as a result of successive garbage collections.

Keywords: LISP machine; garbage collection; cache; free list

1. Introduction

The Artificial Intelligence (AI) field has seen the development of new applications in the last 10 years. Among the AI languages, LISP is the most ancient, applied and known. But conventional Von Neumann machines, oriented to procedural languages like PASCAL and MODULA-2, are inefficient for LISP programs. Research efforts have been developed throughout the world to build appropriate machines for the fast execution of LISP programs.

At the Electrical Engineering Department of the Catholic University of Rio de Janeiro (DEE-PUC/RJ), several studies and prototypes have been developed around the SECD architecture. This machine, proposed by Landin [4] and extended by Henderson [3], was the basis for the first virtual machine implementation, the LispPUC

system [2]. After that, extensions for object oriented programming (LispObj) [9] and for logic programming (LispLog) [1] were developed. A concurrent LISP machine based on the SECD machine with shared memory and a concurent garbage collector was also object of study [5] [6]. Beyond that, a hardware microprogrammed implementation of the original SECD machine was developed with a parallel garbage collector [7] [8].

LISP programs present few logical and arithmetical operations and many memory accesses. The performance of LISP machines are much more dependent of the memory read and write response time than the conventional machines.

Cache memories are an economical and effective way to decrease the memory access mean time in conventional systems. Numerous studies were produced to determine the ideal structure for a cache memory. All these works were based on conventional machines and do not necessarily apply to LISP machines.

A research work to study structures and manegemment strategies of cache memories for LISP machines has begun at the DEE-PUC/RJ. This paper presents the first results of this research.

The SECD machine, implemented as a virtual machine in previous works at the DEE-PUC/RJ, was the basis for this study. The next section presents a résumé of the SECD machine, essential for the comprehension of the analysis that follows: the influence of the free list structure, the cons cell allocation strategies and the importance of the garbage collection on the performance of a conventional cache arquitecture.

2. The SECD Machine

The S.E.C.D. machine is defined basically by 4 (four) registers, a free list and the control part (see figure 1). The four registers **s**, **e**, **c**, **d** behave as pointers to lists and are named **Stack**, **Environment**, **Control List** and **Dump**. The **Stack** is used to hold intermediate and final results as the machine evaluates the expressions. The **Environment** holds the binding of variables during the computation of expressions. The **Control List** holds the program in the machine language during the execution. Finally the **Dump** saves the values of the four registers during a call to a new function.

The **ff** register points to the free list, known as the list of available cells. The machine state is completely defined by the contents of these four registers. Each

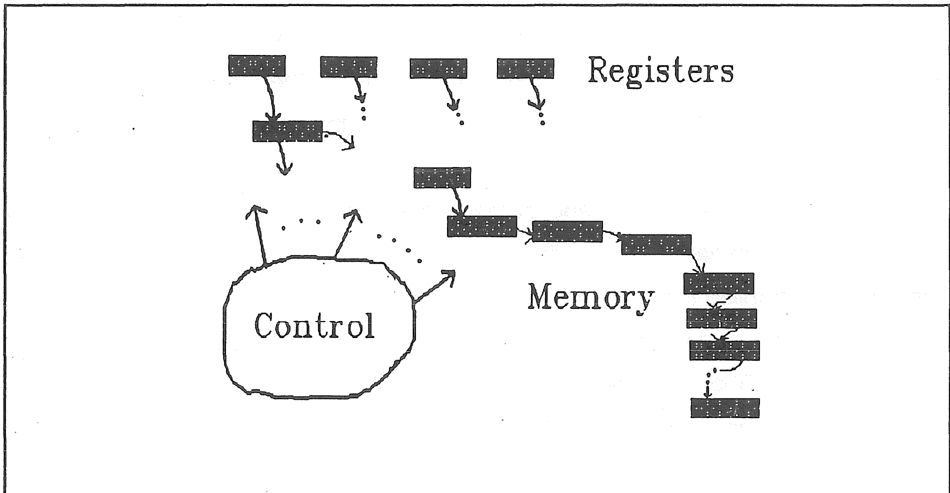

Figure 1: *Simplified description of the SECD machine.*.

machine instruction is defined by the machine state before and after the execution of that instruction.

instruction

s e c d ——————————> s' e' c' d'

The machine control (or the operational semantics) of this machine is defined by the set of all these state transitions. For each instruction, operator or basic LISP function defined in the LISP compiler, there is one (or more) correspondent instruction in the SECD machine.

The machine is organized as follows: the **Control List** points to the program in the machine code. The **Stack** points to the list of arguments. The other registers point to **nil**. The execution of a program in the SECD machine corresponds to a sequence of state transitions defined by the instructions in the **Control List**.

3. 0 Analyses of Access Locality in the SECD Machine

The garbage collection process is typical of LISP oriented systems. During the execution of a LISP program, lists are created and they remain in memory even after becoming useless for the active programs. The garbage collector recovers memory cells which contain the useless parts of the lists, making them available to new lists to be created. In sequential systems the program execution is suspended during the garbage collection. There are parallel LISP systems where program and garbage

collection can be executed concurrently. This work is limited to sequential systems.

The garbage collector produces a chained list - the free list - containing all free memory cells. On demand a cell is taken from the top of the free list. This procedure is repeated until the list runs out of free cells, causing the garbage collection to be re-executed.

Many cache structures and garbage collection policies were studied in the context of this research. Trace driven simulation was the performance analyses methodology applied throughout this work.

Four test programs were used in this study: a LISP Compiler, Tower of Hanoi, Queens and a Fibbonacci Series Generator. Traces were registered for different conditions. In all cases, they were taken during a period without intervening garbage collections. The traces have between 290,000 and 350,000 references.

3.1 The Free List Organization

The simulation results indicated that the cache performance heavily depends on the way the free list is organized in memory. To analyze this problem, two alternatives were considered:

linear free list: the nearer a cell is to the top of the list, the higher is the memory address where it is stored. In a dual organization, the free list starts at a lower address and extends to increasing addresses. It will be clear later that, for the purpose of this work, both organizations are equivalente. This is the most usual way to organize the free list in memory.

random free list: this organization is equivalent to spreading the list cells in a random way along the whole available memory. This organization is only considered with the aim to estimate the effect of the free list organization over the cache performance.

Both organizations were implemented in the virtual SECD machine and their corresponding traces were taken. Figure 2 shows the results from many simulations of a 8 Kbytes, two way set associative cache, for each trace. The curve for the linear list presents lower values than the curve for the random list. It is worth to note that the difference between these two curves becomes greater as the line size increases. This figure shows that the total number of misses can be two times greater for the random

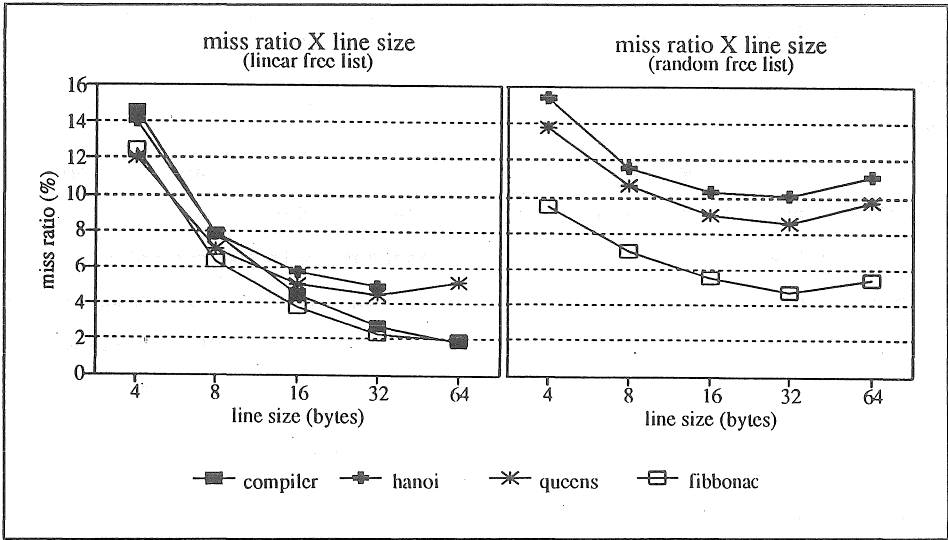

Figure 2 *Miss ratio as a function of line size, for a 8 Kbytes, two way set associative cache using write back, LRU for a a) linear b) random free list.*

free list. The significant miss ratio reduction obtained by doubling the line size clearly indicates that the linear free list enforces the spatial access locality.

These results have motivated a deeper analyses of the linear free list structure. An additional function was introduced in the virtual machine. This function makes a snapshot of the memory soon after the garbage collection is executed. The snapshots save the addresses of each free cell and its successor. It was verified that in more than 94% of the cases, consecutive list cells are stored in consecutive memory addresses. It was also observed that the number of cells stored in memory between two consecutive free list cells are in more than 99% of the cases not greater than 1. Due to this characteristic, the lists created during program execution tend to occupy a contiguous memory region. Therefore accesses to those lists are concentrated in a restricted memory area, in the sense of the spatial locality.

3.2 Allocation of the cons cells in memory

Relative to other cache organizations, direct mapped caches present higher miss ratios. However, their simplicity contributes to reducing the hit circle in the cache. Simplicity and the consequent speed make the direct mapping an interesting cache organization for real systems.

At a certain point of the research the simulations presented very high miss ratios (above 50%) for direct mapped caches. For the same conditions, a two-way set associative cache produced much lower miss ratios. This behavior is not observed in caches for conventional systems [13].

After many experiments changing the cache structure and the memory allocation strategy, the reason for these results became clear.

The following discussion is limited to the conventional list representation in LISP. Each cons cell consists of two fields, car and cdr, each one pointing respectively to the left and right list continuations. Alternative list structures for LISP are proposed in [10][11]. It is assumed that the cache address mapping function is bit selection [12].

Consider the following cons cells allocation. Two vectors, CAR[i] and CDR[i], are created in memory to store the car and cdr pointers. Each vector occupies a contiguous memory area starting respectively at the addresses *base_car* and *base_cdr*. Assuming that each vector element is one word long, the pointers CAR[i] and CDR[i] are stored respectively in the addresses *base_car + i* and *base_cdr + i*, for all i in the considered range.

Taking into account the discussion of the last section about linear free lists, one concludes that the content of CAR[i] and CDR[i] in most cases are equal to 1-α, where α is a natural number greater than zero and near to 1. This implies in a high probability that CAR[i] and CAR[CAR[i]] are stored in the same memory block. A similar statement applies also to CDR[i] and CDR[CDR[i]], to CDR[CAR[i]] and CDR[CDR[i]], and so on.

A simple inspection of LISP programs shows that walking through lists (list traversal) is a very frequent operation. Consider for example the path defined by the following function calls: car(cdr(car(i))). Assuming the described allocation, this operation can be executed just by accessing CAR[i], CDR[CAR[i]] and CAR[CDR[CAR[i]]], in this order.

According to the previous discussion, CAR[i] and CAR[CDR[CAR[i]]] probably belong to the same memory block. An unlucky choice for *base_car* and *base_cdr* may imply that the block containing CDR[CAR[i]] and the block containing CAR[i] and CAR[CDR[CAR[i]]] map into the same associative cache set. In a direct mapped cache, the second access (to CDR[CAR[i]]) will be a miss and the block containing CAR[i] will be displaced from the cache. The third access (to CAR[CDR[CAR[i]]]) will

be also a miss, and the cache will discharge the block containing CDR[CAR[i]] to make room for the new requested block. This can be seen as a competition between a block containing car-pointers and another block containing cdr-pointers for the same cache line. This causes a cache thrashing.

SET SIZE	TEST PROGRAMS											
	Compiler			Hanoi			Queens			Fibbonacci		
	WC	DT	CC	WC	DT	CC	WC	DT	CC	WC	DT	CC
1	49.5	6.1	4.1	49.7	9.7	6.5	47.7	10.9	6.4	53.4	4.3	3.9
2	5.9	2.8	2.3	11.3	7.4	4.4	10.7	8.5	4.4	4.4	3.1	2.7
4	3.5	1.7	1.6	8.5	5.8	3.8	8.7	7.7	3.0	3.3	2.8	2.5
8	1.6	1.6	1.6	6.4	5.2	2.5	8.1	7.4	2.6	3.0	2.8	2.5
16	1.6	1.6	1.6	5.8	4.4	2.2	7.9	6.9	2.2	2.9	2.8	2.5

Tabel 1: *Miss ratio for three different allocations of the cons cells for a 8 Kbytes, four way set associative cache with 32 bytes lines, write back and LRU.*

Table 1 shows, in the WC (worst case) columns, the miss ratios for this allocation strategy. Direct mapped caches have very poor miss ratios. On the other hand, if the set size is doubled, the miss ratio reduces dramatically for all the test programs. It should also be noted that any further doubling of the set size does not produce a such dramatic miss ratio reduction. It is easy to understand that the appointed cause for the cache thrashing does not exist for set sizes greater than 1.

If the addresses *base_car* and *base_cdr* map to sets which stay apart from each other in the cache, the behavior changes completely. The DT (distributed) columns of table 1 show the miss ratios for such allocation. Direct mapped caches have, for this allocation, a much better performance.

A third allocation strategy was considered. It consists of creating an unique vector of cons cells. The car and cdr pointers of each cons cell are stored in consecutive memory addresses. Clearly the competition between accesses to car an cdr does not exist in this case. The CC (car-cdr) columns of table 1 show the miss ratio for this allocation. For all test programs, cache performs better for the CC allocation. Specially for the test programs Queens and Tower of Hanoi, the miss ratio is much lower than for the other allocation strategies considered.

3.3 The Effect of Successive Garbage Collections over the Cache Performance

The lists created during program execution are temporary. Soon after the execution of a program finishes, all memory cells containing such lists are garbage to be recovered by the next collection. Garbage collection practically cancels the effects of the executed programs over the set of free memory cells.

Among the cells that are not collected, are those where the lists pointed by the four machine registes are stored. One could expect that these non collected cells would have a cumulative degradation effect over the miss ratios. The simulation results do not confirm such expectation.

The values presented in figure 1 correspond to traces taken soon after the machine has started its operation. The CC allocation strategy were used in this case. No more than 2 garbage collections has been executed before the traces were taken.

TEST PROGRAM	LINE SIZE (BYTES)									
	4		8		16		32		64	
	*	**	*	**	*	**	*	**	*	**
Compiler	14.6	14.7	7.8	8.0	4.4	4.6	2.7	2.8	1.8	1.9
Hanoi	14.0	14.0	7.9	8.0	5.7	5.6	4.9	4.7	5.4	5.3
Queens	12.1	12.8	7.1	7.4	5.0	5.3	4.4	4.7	5.2	5.3
Fibbonacci	12.5	10.8	6.4	5.6	3.8	2.9	2.3	1.6	2.0	1.2

Tabel 2: *Miss ratio for a 8 Kbytes, two way set associative cache with 32 bytes lines, write back, and LRU for *) few garbage collections **) after many garbage collections.*

To determine how much the garbage collection could affect the cache performance, traces of the same test programs were registered after 15 till 30 garbage collections. Table 2 shows the miss ratios obtained through simulation. Linear free lists were used in both cases. The difference in performance is little. For the Fibbonacci Series Generator the several garbage collections caused even lower miss ratios.

Although the garbage collection affects the access locality and the cache miss ratios, this effect is not cumulative and is limited with a small range of values.

4. Conclusions

This work has demonstrated that the structure of the free lists can have an important impact on the access locality in a LISP machine. Linear free lists enforce access locality and positively affect the cache miss ratios.

The performance of direct mapped caches strongly depends on the allocation of the cons cells in the memory. The option to build two separate vectors, one for the car- and another for the cdr-pointers, may cause cache thrashing. This problem can be avoided by an appropriate choice of the initial addresses of both vectors in memory. The simulations indicated that an allocation where the car- and cdr- pointers of the same cons cell are stored at consecutive addresses, performs better then the other allocations considered.

Finally, the simulations also showed that the successive garbage collections do not cause a cumulative degradation of cache performance.

Many other aspects related to the cache structure for LISP machine are still to be studied. Memory actualization, prefetch and replacement strategies, are some examples. In the context of parallel LISP machines there is a lot of other open questions.

Acknowledgements: The authors want to acknowledge the support of CNPq and Secretaria de Ciência e Tecnologia of the Brazilian Government.

5. Bibliography

[1] Corbucci, Dante. "LispLog: Uma Linguagem para a Programação Funcional e para a Programação em Lógica". M. Sc. Dissertation, Electrical Engineering Department, PUC-RJ, 1989.

[2] Guidacci da Silveira, G. F., "Uma Primitiva Não-Determinística para o LISP e sua Aplicação na Programação por Retrocesso," Simpósio Brasileiro de Inteligência Artificial (3º), Rio de Janeiro, 1986, pp. 236- 245.

[3] Henderson, P., "Functional Programming - Application and Implementation," Prentice-Hall International, 1980.

[4] Landin, P.J, "The Mechanical Evaluation of Expressions," Computer Journal, vol 6, no.4, 1963, pp. 308-320.

[5] Martins, W. S., "Um Sistema LISP Paralelo: Concepção e Simulação," M. Sc. Dissertation, Electrical Engineering Department, PUC/RJ, 1990.

[6] Martins, W. S. e Guidacci da Silveira, G. F., "Um sistema LISP Paralelo," Simpósio Brasileiro de Arquitetura de Computadores e Processamento Paralelo (III), Rio de Janeiro, 1990, pp. 238-251.

[7] Oliveira, F. S. G., "Compilador de Microcódigo para uma Máquina S.E.C.D.," Final Year Project, Departamento de Engenharia Elétrica, PUC/RJ, 1987.

[8] Oliveira, F. S. G., "ENTERPRISE, Uma Arquitetura LISP," M. Sc. Dissertation, Electrical Engineering Department, PUC/RJ, 1991.

[9] Silva, R. P., "LispObj: Uma Linguagem LISP com Extensões para a Orientação a Objetos," M. Sc. Dissertation, Electrical Engineering Department, PUC/RJ, 1989.

[10] Pleszkun, A. R., "An Architecture for Efficient Lisp List Access," Int. Symp. on Computer Architecture, 1986, pp.191-198.

[11] Potter, J. L., "Alternative Data Structures for List in Associative Devices," Proc. Int. Conf. Parallel processing, 1983, pp. 486-491.

[12] Smith, A. J., "Cache Memories," Computing Surveys, Sept. 1982, pp. 473-530.

[13] Smith, A. J., "Cache Evaluation and the Impact of Work Load Choice," Int. Symp. on Computer Architecture, 1985, pp. 64-73.