Functional Languages: A Performance Study

S. Mansoor Sarwar
Department of Electrical Engineering, Multnomah School of Engineering, University of Portland, Portland, Oregon

Marwan H. Abu-Amara
Department of Electrical Engineering, Texas A & M University, College Station, Texas

This article describes a study evaluating the run time behavior of two functional languages, combinator-based SASL and environment-based Franz LISP, for a set of algorithms. The idea was to measure the effectiveness of the instruction set of a conventional processor and Turner's combinators as the instruction set for a processor that runs functional languages. The study shows that, statistically, the combinator-based implementation of SASL is better than the environment-based implementation of Franz LISP in space and time for at least small to medium-size input data sets.

INTRODUCTION AND PROBLEM STATEMENT

An implementation technique for functional languages that has received considerable attention in recent years is combinator-based graph reduction [1-5]. This technique was first introduced by Turner [3] in his implementation of a purely applicative language called SASL. In the SASL environment, the bound variables are abstracted (removed) from expressions and replaced by a number of constants called combinators. The resulting variable-free expression is then complied into a general digraph which is progressively reduced at run time until it is no longer reducible, and is called the normal form (answer) of the given expression.

The main focus of this article is to describe a study which was conducted to find out how environment-based implementation of functional Franz LISP [6-8] and combinator-based implementation of SASL compare with each other for a set of algorithms. Our study primarily focused on collecting the following statistics:

- The static code sizes, i.e., the number of memory words needed to store the compiled codes for a set of abstract data types (ADTs) and small to medium-size programs for SASL and Franz LISP.
- The ratio of the number of memory references made by SASL and LISP versions of the programs for small to medium-size input data sets.

These statistics established the space and time behaviors of environment- and combinator-based graph reduction models for a set of representative algorithms. They can be used to design a processor instruction set that efficiently supports the execution of functional languages.

In this study, Turner's combinators were assumed to comprise the instruction set of a hypothetical processor. The number of memory references made for SASL programs was based on the assumption that SASL programs are executed on such a processor. This assumption is realistic because, for our test programs, the average number of arguments needed for a VAX 8350 instruction and a Turner combinator came out to be about 1.67 and 2.65, respectively. Also, the number of arguments for the Turner combinators used in the SASL environment varies between 1 and 4, which is very similar to the range of arguments needed by the instructions for a typical conventional processor. To make our analysis as fair as possible, we implemented the same algorithms in both language environments. We tried to use similar language constructs and functions (primitive as well as nonprimitive) in both languages. In addition, the overhead code generated as part of the compiled

Address correspondence to Professor S. Mansoor Sarwar, Dept. of Electrical Engineering, Multnomah School of Engineering, University of Portland, 5000 N. Willamette Blvd., Portland, OR 97203.

655 Avenue of the Americas, New York, NY 10010
0164-1212/93/$6.00

version of a LISP program was not considered while calculating the number of memory references made by the program during its execution.

PERFORMANCE EVALUATION METHODOLOGY

To obtain the required statistics, the SASL run time system was tailored to get execution time data for a set of representative ADTs and small to medium-size programs. Our modified system lets us view the initial combinatory graph, the combinatory graph after every reduction step, the initial combinatory string, the combinatory string after each reduction step, the contents of a range of graph nodes, and total and percent usage of each combinator for a given program execution.

To collect statistics for the LISP counterparts of the SASL programs, the Franz LISP compiler Liszt was used to generate assembly versions of source programs. The assembly code for each program was then passed through a filter that gave its static code size as output. Since we did not have the source code for Liszt, a set of counters were placed at appropriate places in a LISP program to calculate the frequency of execution of each function in the program. These instruction frequencies were then processed to calculate the number of assembly language instructions executed and the number of memory references made for the given program.

RESULTS AND DISCUSSION

In this section, we analyze the statistics obtained for a few of our benchmark programs, including symbolic differentiation, matrix multiply, and various searching and sorting algorithms. We also describe the relative performance of various sorting and searching algorithms in SASL and Franz LISP environments.

Table 1. Static Code Sizes For SASL And Franz LISP Programs

Program	LISP (VAX 8350 Instructions)	SASL (Turner Combinators)	Ratio (LISP/SASL)
Symbolic differentiation	6629	2382	2.783
Matrix multiply	628	67	9.373
Sorting			
Insertion	170	54	3.148
Quick	324	59	5.492
Tree	406	134	3.030
Searching			
Sequential	110	31	3.548
Binary	517	95	5.442
Tree	221	63	3.508
Maketree	155	88	1.761
Tree traversal			
Inorder	63	30	2.100
Preorder	65	31	2.097
Postorder	73	34	2.147

The statistics taken for the benchmark programs have shown that the static code sizes for SASL programs are always smaller than their LISP counterparts. Table 1 shows the static code sizes for the various benchmark programs and ADTs we analyzed. The table clearly shows that the amount of memory needed to store the combinatory code version of a program is always smaller than the amount of memory needed to store the assembly code version of the same program. The ratio of the static code sizes for the matrix multiply program is outside the normal range. The primary reason for this abnormality is that SASL has a more powerful library function "map" (the most heavily used function in our matrix multiply algorithm) than its Franz LISP counterpart. Therefore, we wrote our own map function to ensure that we used the same algorithm. However, this increased the static (hence dynamic) code size of the LISP version of matrix multiply.

As for the average number of memory references made during the execution of different programs, the LISP version of the symbolic differentiation program made about 15% more memory references for differentiation of various functions than its SASL counterpart. That is, on the average, the combinator-based version of the symbolic differentiation program ran 15% faster than the environment-based (standard assembly code) version. For matrix multiply, the speedup is indicated by the curve shown in Figure 1. This curve shows that speedup decreased with increase in size of the input matrices and that the speedup became almost constant for large matrices. In fact, the speedup became almost a constant 1.4 for matrices of size ≥ 10.

Figures 2 and 3 show the speedup comparisons for various sorting and searching algorithms in worst-case scenarios. The curves in Figure 2 show that the combinatory code executed faster than the normal assembly code for all three sorting algorithms. Although speedups decreased quickly with increase in the size of input list, they seemed to become constant for list lengths of > 80. The curves also show that speedup was largest for insertion sort, followed by tree sort and Quick sort, respectively.

The curves in Figure 3 show that among all the searching algorithms we analyzed, the combinatory version of sequential search gave the highest speedup over its assembly code counterpart. In addition, this speedup remained a constant two, whereas the speedup for binary and tree search algorithms decreased with increase in input list size. Analysis also

J. SYSTEMS SOFTWARE
1993; 20:163–167

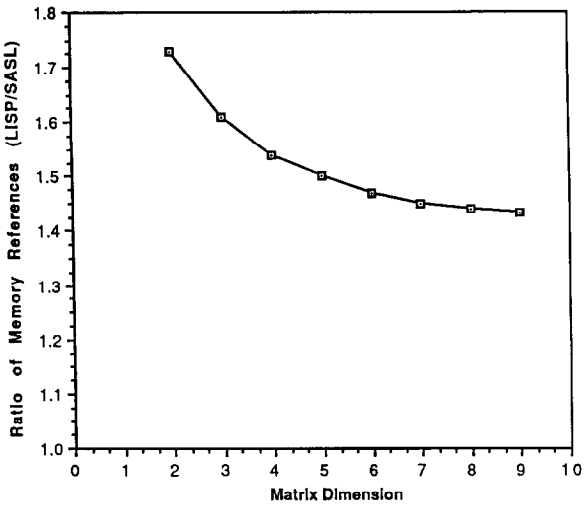

Figure 1. Performance of matrix multiply.

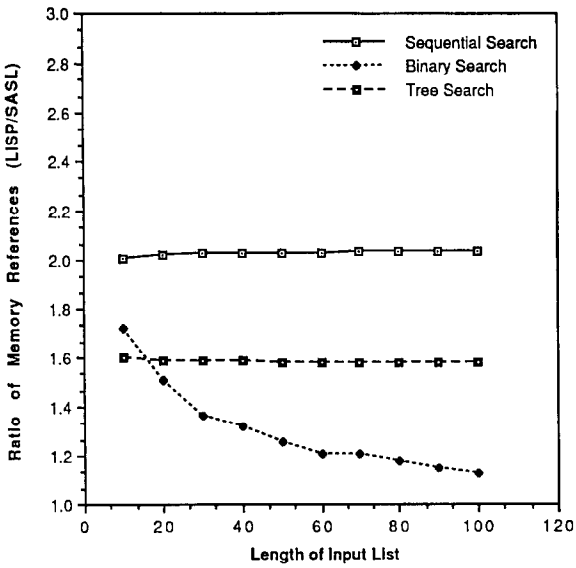

Figure 3. Performance of search algorithms.

showed that the speedup ratio for tree sort decreased by about 0.01 for every 100-element increase in the size of the input list. Furthermore, among the searching algorithms, sequential search was the most efficient, both in space and time, in combinatory as well as assembly versions.

For searching and sorting algorithms, the number of memory references needed to execute the recursive portions in the assembly versions was a little larger than the number of memory references needed to execute the corresponding portions in the combinatory codes. Furthermore, for binary search and all sorting algorithms, the nonrecursive portions of assembly codes were a little larger than the corresponding combinatory codes. Therefore, as the size of the input list increased, the relative significance of the nonrecursive portions decreased, thereby leveling the speedup curves. For sequential search and tree search, however, the ratio of memory references was almost constant, both for recursive as well as nonrecursive portions of the codes. Therefore, the speedup remained almost constant for these algorithms.

The theoretical complexities of sequential, tree, and binary search algorithms are $O(n)$, $O(\log n)$, and $O(\log n)$, respectively. However, actual implementation of these searching algorithms has shown that binary search is the worst in time behavior, followed by tree search and sequential search. Figures 4 and 5 further show that sequential search was

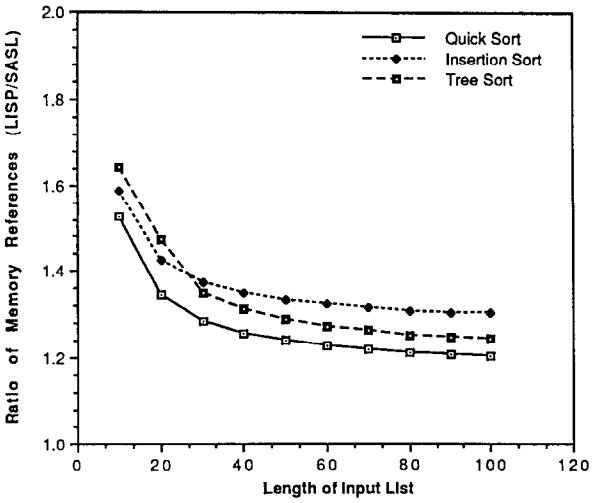

Figure 2. Performance of sort algorithms.

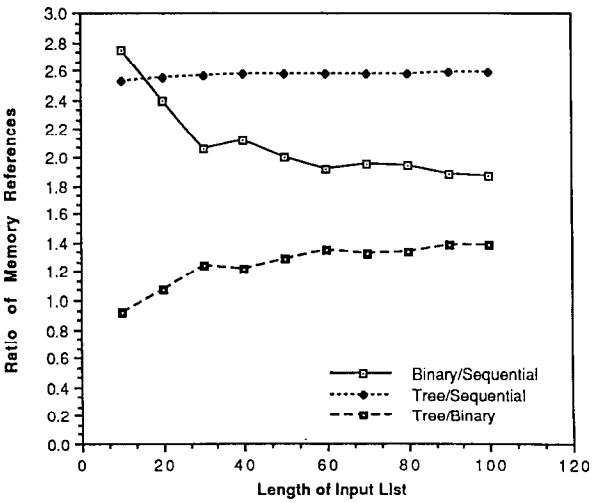

Figure 4. Performance of SASL search.

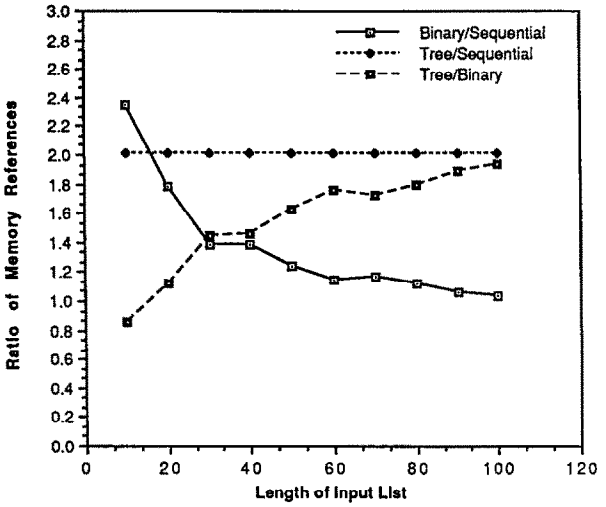

Figure 5. Performance of LISP search.

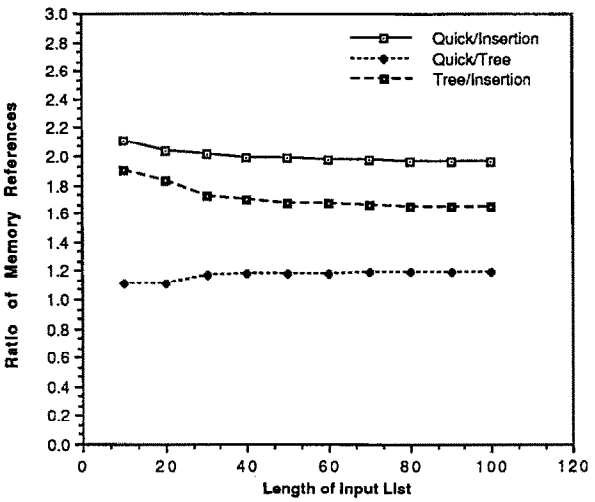

Figure 7. Performance of LISP sort.

about 2.5 and 2 times faster than tree search in SASL and Franz LISP environments, respectively. Similarly, sequential search behaved better than binary search for list sizes of ≤ 100 elements in Franz LISP. In SASL, sequential search continued to behave much better than binary search for lists of > 100 elements. Tree search performed better than binary search for list sizes of ≤ 17. For lists of > 17 elements, binary search behaved better than tree search.

Among the sorting algorithms we analyzed, insertion sort performed about twice as well as quick sort and about 1.6 times as good as tree sort (Figures 6 and 7). Performance was a little better in the SASL environment. Tree sort performed about 1.2 times as well as quick sort in both environments for the range of list sizes that we considered.

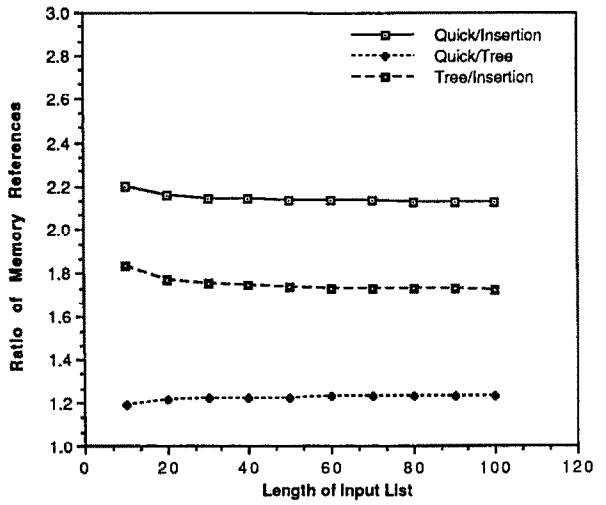

Figure 6. Performance of SASL sort.

CONCLUSIONS AND FINAL REMARKS

The study shows that for small to medium-size programs and input data sets, compiled functional languages will execute faster on a processor whose instruction set is the combinators Turner used for implementing SASL, as opposed to a conventional processor. The behavior of a wide range of regular, irregular, symbolic, and nonsymbolic programs in combinator-based SASL and environment-based Franz LISP (on VAX 8350) has clearly demonstrated this. Furthermore, among searching algorithms, sequential search behaved the best, followed by binary and tree search. Among sorting algorithms, insertion sort performed the best, followed by tree and quick sort.

The study further shows that the behavior of an algorithm in a given programming language depends on the data domains available in the language and their implementation. If data domains are sequential, then sequential algorithms perform better than nonsequential (tree, divide-and-conquer, etc.) algorithms. If data domains are nonsequential, then nonsequential algorithms behave better than sequential algorithms. Since SASL and Franz LISP have list as one of their heavily used data domains, sequential algorithms perform better than nonsequential algorithms if list is used as the fundamental data structure. This behavior is clearly reflected by the curves

illustrated in Figures 4–7 for a number of searching and sorting algorithms.

To further the research described here, a set of benchmarks are being analyzed to find out most often used Turner combinators and combinator strings. These statistics can be used to design a processor with most heavily used Turner combinators as its instruction set. Also, some of the most often used combinator strings will be substituted by equivalent but smaller combinatory strings or new families of combinators. The main focus with respect to the new combinator families will be on list-manipulation combinators, because our preliminary study has shown that list manipulation is the major bottleneck in functional languages. The reason for this bottleneck is that most functional languages have list as one of their most heavily used compound data domains, but use such primitive list-manipulation combinators as cons, car, and cdr. We will focus on designing a set of combinator families to efficiently support higher level list operations, along with a new representation of list to support these combinator families. In so doing, we hope to be able to speed up vector- and array-like operations by allowing random access within a list as well as concurrent operations on subsets of list elements.

REFERENCES

1. S. K. Abdali, An Abstraction Algorithm for Combinatory Logic, *J. Symbol. Log.* 41, 222–224 (1976).
2. R. J. M. Hughes, Super-Combinators: A New Implementation Method for Applicative Languages, *Proceedings of the ACM Symposium on LISP and Functional Programming*, Pittsburgh, Pennsylvania, 1982, pp. 1–10.
3. D. A. Turner, A New Implementation Technique for Applicative Languages, *Software-Practice and Experience* 9, 31–49 (1979).
4. D. A. Turner, Another Algorithm for Bracket Abstraction, *J. Symbol. Log.* 44, 267–270 (1979).
5. D. A. Turner, *SASL Language Manual*, University of Kent, Canterbury, U.K., 1983.
6. J. F. Foderaro, K. L. Sklower, and K. Layer, *The Franz LISP Manual*, University of California, Berkeley, California, 1983.
7. J. F. Foderaro, *The Franz LISP System*, University of California, Berkeley, California, 1983.
8. R. Wilensky, *LISPcraft*, W. W. Norton, New York, 1986.