Comput. Lang. Vol. 19, No. 3, pp. 205–211, 1993
Printed in Great Britain. All rights reserved

0096-0551/93 $6.00 + 0.00
Copyright © 1993 Pergamon Press Ltd

RUN-TIME BEHAVIOR OF SASL PROGRAMS: A PERFORMANCE STUDY

S. MANSOOR SARWAR
Multnomah School of Engineering, University of Portland, 5000 N. Willamette Blvd, Portland, OR 97203-5798, U.S.A.

(Received 19 June 1992; revision received 10 November 1992)

Abstract—This paper describes the results of a study to measure the performance of various searching and sorting algorithms, and the combinatory behavior of the SASL (St. Andrew's Static Language) run-time system for a wide range of algorithms, abstract data types and programming styles. The study shows that sequential algorithms perform much better than non-sequential algorithms in the SASL environment. The study also shows that there are families and strings of combinators that are used frequently. Optimization of such combinators promises an efficient sequential reduction machine based upon the combinator model.

functional programming languages combinator graph reduction performance evaluation searching and sorting algorithms

1. INTRODUCTION AND PROBLEM STATEMENT

An implementation technique for functional languages [1–3] that has received considerable attention in recent years is the combinator-based graph reduction model, first introduced by Turner in his implementation of a purely applicative language called SASL [4, 5]. In the SASL environment, bound variables are *abstracted* (removed) from expressions and replaced by constants called *combinations* [6]. The resulting variable-free expression is then compiled to a digraph which is progressively reduced at run-time until it is no longer reducible and is called the *normal form* of the given expression.

The main focus of this research was to obtain answers for the following questions:

1. How do different sorting and searching algorithms behave in the SASL environment?
2. What are the most frequently used combinators in the SASL environment?
3. What is the percentage usage of each Turner combinator in the SASL environment?
4. What are the frequently used combinator strings in the SASL environment?

The only empirical study found in literature which tries to answer similar questions is by Hartel and Veen [7]. This study gives us answers to questions 2 and 3 for a set of programs which were chosen on *ad hoc* basis and ran with input data of small sizes. We think that this approach gives us a sample space which is small in size and limited in nature and, therefore, does not give us the true average behavior in the statistical sense. In order to make our statistics more meaningful, we selected a wide range of often-used abstract data types, algorithms and programming styles as our benchmarks. Our programs range from small to medium in code size.

To fulfill our goals, the SASL run-time environment was tailored to get the needed execution-time data. Our modified system lets us view the initial combinator graph, combinator graph after each reduction step, initial combinator string, combinator string after each reduction step, contents of a range of graph nodes, and total and percentage usage of each combinator for a given program execution. The behavior of the benchmarks was observed by carefully examining them before, during and after their evaluation for small to medium size data. Whenever possible, the programs were made to consume considerable amount of CPU time by running them with input data of appropriate size and shape. Large and adequately arranged input data supplied to matrix multiply, searching and sorting programs forced hundreds of thousands (sometimes more than a million) of combinator reductions. These many reduction steps sometimes took over 20 min of user time on department's VAX 8350 system with average load.

The behavior of searching and sorting algorithms was observed for worst-case scenarios by providing them with appropriately-ordered input data. For other programs, we studied the average case behavior in the practical sense. For example, for the symbolic differentiation program, all possible expressions, ranging from a constant to logarithms, were considered. Similarly, for the database program, the behavior of all possible operations on a database, like inserting a record, deleting a record, searching for a record, etc., was observed.

To answer the above questions, we collected the following data:

1. The estimated number of memory references made by different searching and sorting algorithms for small to medium size data.
2. The number of reductions needed to reduce expressions involving operations defined on data types under investigation.
3. The percentage usage of each combinator for a given program run.
4. The commonly occurring patterns of expressions and associated combinator strings.

In addition to answering our questions, these statistics helped us establish the relative importance of each combinator and gave us insight into formulating new and more optimal combinators for a combinator-based functional language.

The abstract data types that were investigated are stack, queue, tree, array, record and matrix. Operations on these data types included sorting, searching, tree construction, tree traversal, array construction and initialization, record construction, etc. Everyday and real-world programs that were analyzed include matrix multiply program, a small database manager and symbolic differentiation program.

2. RESULTS AND DISCUSSION

In this section, we discuss the results of our study. First we describe the behavior of the searching and sorting algorithms, and then answer the questions regarding the combinatorial behavior of our benchmarks.

2.1. Behavior of searching and sorting algorithms

Hartel and Veen [7] assume that the number of reduction steps are representative of the computational complexity of a program and that graph rewrite due to any single combinator consumes one unit of time. In our opinion, these assumptions are not realistic and, therefore, may not be used while comparing the performance of algorithms. Clearly, the complexity of a reduction step varies from one combinator to another. For example, the reduction of *S1* combinator is more expensive than the reduction of either *I* or *K* combinator due to the number of nodes read, created and written for each of the three combinators. Similarly, the reduction of *B1* is more expensive than the reduction of either *B* or *C* combinator.

We think that the best comparison is in terms of the number of memory references made by each algorithm for a given input data size. However, since the SASL run-time system compiles every program in a general combinator digraph, it was not possible for us to find out the actual number of memory references for a given program run. We, therefore, devised a model for algorithm comparison.

2.1.1. Model used for estimating the performance of algorithms. In the SASL environment, each graph node comprises of three fields: a tag field, a left subgraph field and a right subgraph field. In our model, we assign a positive integer to each Turner combinator which represents its reduction complexity in terms of the number of nodes read, created and written. The weight assigned to a combinator in our model is equal to $N_{arg} + 4N_{nn}$, where N_{arg} is the number of arguments of the combinator and N_{nn} stands for the number of new nodes created during the reduction of the combinator. The factor of 4 accounts for the creation of one new node and assignment of values to its three fields. Table 1 shows weight assignments to Turner combinators.

We calculate the *weighted sum* for all combinator reductions for a given program run. This process is repeated for different input data sizes for all algorithms. Finally, these weighted sums are used to compare the performance of algorithms under consideration. The following formula is used to calculate the weighted sum for a given program run:

Table 1. Weights of Turner's combinators

Combinator	Weight	Combinator	Weight
I	1	FDIV	7
TL	2	MOD	7
Y	3	POWER	7
HD	6	MATCH	8
NOT	6	COND	8
NEG	6	U	12
K	7	U_	12
EQ	7	C	13
AND	7	B	13
OR	7	C_p	13
GR	7	B_p	13
GRE	7	S	18
MUCHGR	7	S_p	18
MINUS	7	TRY	18
PLUS	7	CI	19
TIMES	7	BI	19
INTDIV	7	SI	24

$$N_{mr} = \sum_{i=1}^{n} R_i W_i$$

where, N_{mr} is the number of memory references, R_i is the number of reductions for combinator i, W_i the weight for combinator i, and n the number of combinators used for a given program run. These sums are then used to compare the execution speeds of the selected searching and sorting algorithms.

2.1.2. Speedup comparison results. Theoretical complexities of sequential, binary and tree search algorithms are $O(n)$, $O(\log n)$ and $O(\log n)$, respectively. However, the actual implementation of these algorithms in SASL has shown that binary search is the worst in time behavior, followed by tree search and sequential search. Curves in Figs 1 and 2 show the relative performance of the searching and sorting algorithms, respectively.

The curves in Fig. 1 show that sequential search is about 2.6 times faster than tree search.

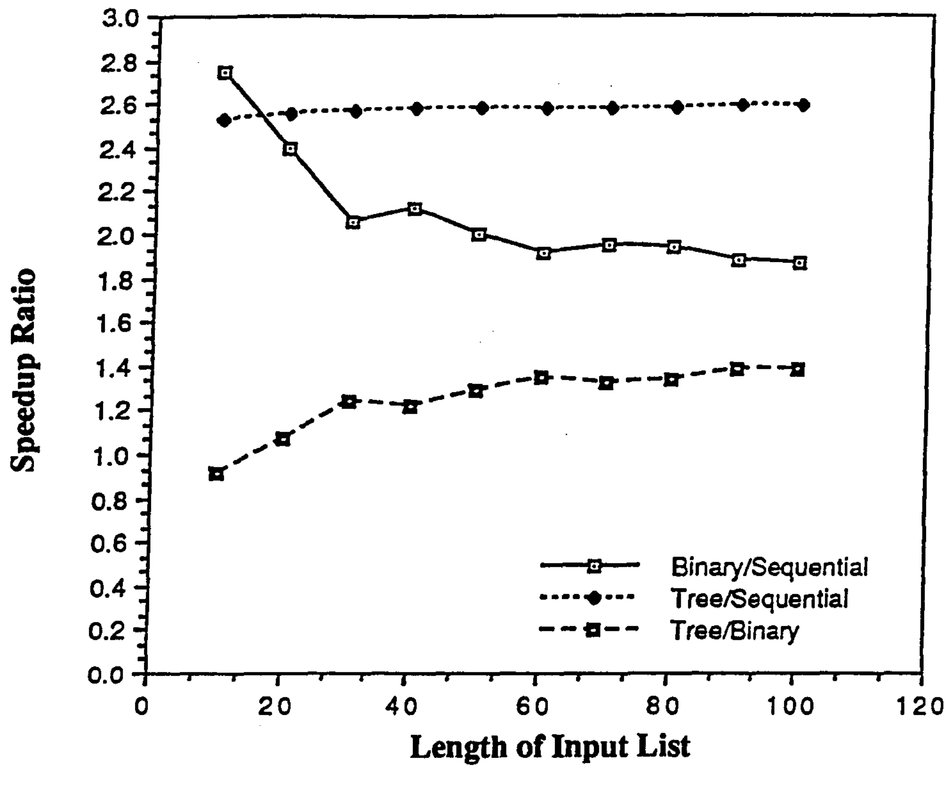

Fig. 1. Performance of searching algorithms.

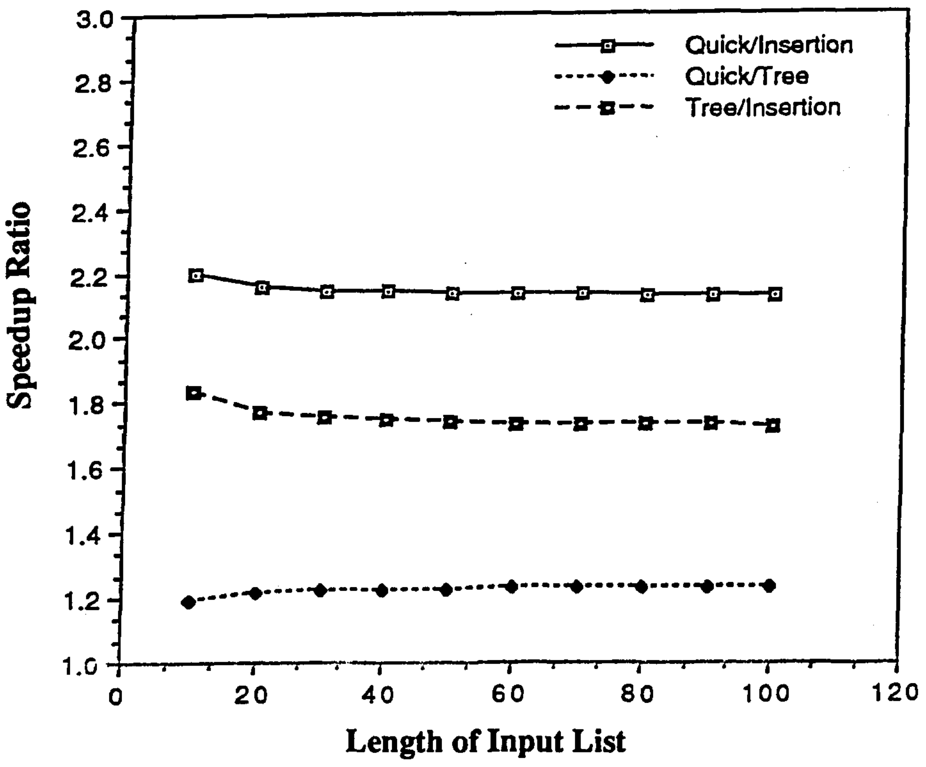

Fig. 2. Performance of sorting algorithms.

Compared to binary search, sequential search is about 2.8 times faster for an input list of size 10. But, as the list size increases, the speedup factor becomes smaller. However, the slope of the speedup curve starts decreasing with increase in the input list size. The speedup becomes almost a constant 1.86 for list sizes of >100. Comparison between tree search and binary search shows that binary search is about 1.4 times faster than tree search for list sizes >100.

The curves in Fig. 2 shows that the most sequential of the three sorting algorithms has the best performance and the least sequential has the worst performance. For example, insertion sort is about 2.1 times faster than Quick sort and 1.7 times faster than tree sort. Furthermore, tree sort is about 1.2 times faster than Quick sort.

2.2. Combinatory behavior of SASL

The results of this study have shown that there are a number of combinator families that are used for certain types of SASL expressions. For example, the matching combinators (MATCH, TRY and U_) are used for pattern matching (a substitute for the case expression in SASL) structures. The B_p, C_p and S_p combinators are used whenever list operator ' : ' is used. The S, B and C combinators are used for expressions which are combinations (e.g. f applied to x). The statistics obtained for different abstract data types, algorithms and programming styles are summarized in Table 2. The percentage usage of the Y combinator is not given here because it was always very small, usually $<1\%$, a fact which is also supported by the study done by Hartel and Veen [7]. Some of these statistics do not truly represent the real-world behavior of the abstract data types and programs evaluated because of the size of input data and programming style adopted by the programmer. They do, however, give a general intuition into the architectural support needed for efficient implementation of functional languages (in particular SASL) based on combinators and graph reduction.

The two most-heavily used combinator families are S_B_C (comprising of S, B, C, S1, B1 and C1 combinators) and Matching (comprising of MATCH, TRY and U_combinators), followed by K_I (comprising of K and I combinators) and List (comprising of S_p, B_p, C_p, HD, TL, U and

Table 2. Abstract data types and percentage usage of combinator families

Abstract data type	Operations		Percentage occurrence of combinator families						
			S_B_C	K_I	List	Matching	Conditional	Boolean	Arithmetic
Stack	Pop			50		50			
	Push		33	33	33				
Queue	Enqueue	Alg1	20	40	40				
		Alg2	14	7.0	79				
	Maketree		56	12	8.1	19	2.4	2.4	
	Tree	In	39		16	45			
Tree	traversal	Pre	42		15	43			
		Post	33	4.7	24	38			
Array	Initialize array		27	9.0	14	37			13
	Reset array		23	10	13	45			9.4
	Add arrays		59	3.8	7.6	26			3.8
Record	Make record		48	12	5.1	28	3.6	3.6	
Matrix	Matrix multiply		23	24	8.2	40		1.5	2.7
Searching	Sequential		42	18	4.6	21	7.0	7.0	
	Binary		32	17	2.9	33	1.2	2.8	9.5
	Tree		59	10	2.0	19	5.2	5.2	
Sorting	Insertion		51	14	6.2	21	3.4	3.6	
	Quick		32	19	13	29	3.5	3.6	
Database manager			38	16	3.5	30	4.6	4.4	2.7
Symbolic differentiation			44	26	2.6	6.7	8.4	9.6	4.4

APPEND combinators). The least used combinators are for evaluating arithmetic and boolean expressions. The average utilization of each combinator family over all our benchmark programs is given in Table 3. Since the nature of our benchmark programs varies from highly regular to non-regular, these numbers are good indicative of the overall utilization of each combinator family.

Table 4 shows a comparison between the percentages obtained by Hartel and Veen [7] and our study for the utilization of various combinator families. This table clearly shows that although these percentages are fairly close for the most part, they do, however, differ considerably for some families. Especially, the numbers obtained by Hartel and Veen do not show high utilization of list manipulating combinators. The primary reason for this disparity between the two studies seems to be the fact that the choice of benchmarks by Hartel and Veen was on fairly *ad hoc* basis, whereas our benchmarks represent a good range of abstract data types, algorithms and programming styles.

A number of frequently occurring combinator strings, such as 'U_B1', 'U_B' and 'C_pI NIL', were also observed. This means that reduction of certain expressions can be optimized by introducing new combinators that perform the same operations as do these combinator strings. However, as this optimization process continues, it soon becomes necessary to abstract more than one variable at a time from an expression. In fact, certain patterns of commonly occurring combinators come about because the abstraction method proposed by Turner is highly recursive and abstracts one instance of a variable at a time.

The study also showed that even simple operations can take a substantial number of reductions. There are two reasons for this:

1. Despite the fact that the most-heavily used data domain in SASL is a list, the system does not have efficient list-manipulation combinators. Since the list constructor operator, ':', is

Table 3. Percentage usage of each combinator family

Combinator family	Percentage usage
S_B_C	36
K_I	16
List	15
Matching	27
Conditional	2.0
Boolean	2.2
Arithmetic	2.3

Table 4. Comparison with percentages obtained by Hartel and Veen

Combinators	Percentage utilization	
	Hartel and Veen	Our study
K, I	27	16
MATCH, TRY, U's	19	27
Logic and arithmetic operators	6.9	4.5
COND	1.8	2.0
S, B, C, S_p, B_p, C_p, HD, TL, Select, APPEND	46	51

an interior node between head and tail, selecting nth element of a list requires n successive reduction steps.

2. The reduction process in the SASL environment is highly recursive and sequential, caused by the recursive abstraction method used.

It is for the second reason that (as has been discussed in section 2.1.2.) sequential search was found to be about 2.6 times faster than tree search and about 1.86 times faster than binary search. Similarly, insertion sort came out to be about 2.1 times faster than Quick sort and about 1.7 times faster than tree sort. The reason for these counter-intuitive results is that much larger combinatory codes are generated for non-sequential programs. Therefore, if one is to implement a functional language on a multiprocessor environment, it becomes imperative to develop operations and combinators that efficiently manipulate the list structure, and the method of abstracting one instance of a variable at a time has to be avoided.

3. CONCLUSIONS AND FINAL REMARKS

The results of the study have shown that the behavior of an algorithm in a combinator-based functional programming language is dependent upon the data domains available in it and the type of combinators used to implement it. Since the fundamental and most-heavily used data domain in SASL is a list and combinators resulting from the recursive abstraction method are well suited to recursive and sequential expressions, sequential algorithms perform much better than non-sequential (tree, divide-and-conquer, etc.) algorithms. In case of more sophisticated data domains and algorithms, the resulting codes become large and, therefore, take more time to execute. For example, as it was mentioned before, sequential methods of searching and sorting were found to be the most efficient in the SASL environment. This behavior is clearly reflected by the curves in Figs 1 and 2.

The study has also shown that there are a set of combinator families that are used frequently. Furthermore, a large percentage of reductions result from pointer manipulations. These characteristics greatly influence the architecture of the underlying graph reduction machine. A sequential combinator-based graph reduction machine should have the following features:

(a) The architecture of the machine should efficiently support pointer manipulation.
(b) The implementation of the combinators of S_B_C, K_I, Matching, and List families should be optimized as much as possible.

For a parallel combinator-based graph reduction architecture, a lambda-lifting technique that abstracts a number of bound variables at a time (like the ones proposed by Abdali [8] and Hughes [9]) should be used because such a technique allows parallel execution of subexpressions quickly. Secondly, a set of operators and combinator families should be designed that allow random access of list elements and can perform concurrent operations on them. Furthermore, a new list representation should be designed to support the semantics of these operators and combinators.

The list-manipulation operators and combinators are of utmost importance because even though Abdali's and Hughes' algorithms allow concurrent execution of functional expressions quickly, list-manipulation operators remain recursive. The idea is to remove the "*functional bottleneck:*" the sequential and recursive list manipulation. It is author's belief that such an approach would improve the execution speed of functional programs tremendously and make them very competitive on parallel machines.

REFERENCES

1. Backus, J. Can programming be liberated from von-Neumann style? A functional style and its algebra of programs. *Commun. ACM* **21**(8): 613–641 (1978).
2. Henderson, P. *Functional Programming: Application and Implementation.* London: Prentice-Hall; 1980.
3. Peyton-Jones, S. L. *The Implementation of Functional Programming Languages.* London: Prentice-Hall; 1987.
4. Turner, D. A. A new implementation technique for applicative languages. *Software-Practice Experience* **9**: 31–49 (1979).
5. Turner, D. A. *SASL Language Manual.* Canterbury, U.K.: University of Kent; 1983.
6. Curry, H. B. and Feys, R. *Combinatory Logic,* Vol. 1. Amsterdam: North-Holland; 1958.

7. Hartel, P. H. and Veen, A. H. Statistics on graph reduction of SASL programs. *Software Practice Experience* **18**(3): 239–253 (1988).
8. Abdali, S. K. An abstraction algorithm for combinatory logic. *J. Symbolic Logic*, **41**(1): 222–224, 1976.
9. Hughes, R. J. M. Super-combinators: a new implementation method for applicative languages. *Conference Record of the ACM Symposium on Lisp and Functional Programming*, pp. 1–10, Pittsburgh, Pennsylvania, U.S.A.; 1982.

About the Author—S. Mansoor Sarwar received his undergraduate degree in Electrical Engineering from University of Engineering and Technology, Lahore, Pakistan and M.S. and Ph.D. degrees in Computer Engineering from Iowa State University. He is currently an Assistant Professor in the Department of Electrical Engineering at the University of Portland, Oregon. Dr Sarwar's teaching and research interests are in programming languages, operating systems, parallel and distributed computing and performance evaluation.